数据结构

23物联网工程 2023124306

第 1 章 绪论

1.1 数据结构的研究内容

数据结构是计算机科学中非常核心的内容之一，研究的是如何在计算机中组织、存储和管理数据。其核心目的是确保数据的高效访问与操作，主要包括以下几个方面：

- **数据的组织方式**：数据结构定义了数据元素的存储方式以及它们之间的关系。例如，数组、链表、树、图等都是不同的数据结构，它们适应不同的应用场景。
- **数据的操作方式**：数据结构不仅仅是存储数据的容器，还是执行各种操作的基础，如插入、删除、查找、排序等操作。
- **算法的实现**：数据结构与算法密切相关，良好的数据结构设计能够让算法实现更高效。不同的操作需要选择不同的数据结构来实现，比如查找操作常用的结构有数组、链表、哈希表、二叉搜索树等。

研究数据结构不仅关注数据的存储，还涉及如何利用数据结构来优化程序的执行效率。因此，数据结构是理解和解决计算机科学各类问题的基础。

1.2 数据结构的基本概念和术语

在学习数据结构时，首先要掌握一些基础的概念和术语。这些基本概念为后续深入学习提供了重要的理论基础。

1.2.1 数据、数据元素、数据项和数据对象

- **数据**：是指一切可以被计算机处理的信息。数据不仅仅限于数字，还可以包括文字、图像、声音等信息。
- **数据元素**：是数据的最基本单位，表示一项具体的、不可再分的数据。例如，在一个学生信息表中，学生的姓名、年龄、学号等都可以看作数据元素。
- **数据项**：是数据元素的具体内容。在上述例子中，数据项包括学生的姓名“张三”、年龄“20”、学号“202101”。
- **数据对象**：是由若干数据元素按照一定的逻辑关系组合起来的整体。比如，学生信息表就是一个数据对象，它包含多个数据元素（如姓名、年龄、学号等）。

这些术语帮助我们理解数据如何在计算机中表达和管理。数据对象可以看作是对数据元素的组织，它们之间通过一定的关系被关联起来。

1.2.2 数据结构

数据结构是指计算机中存储、组织数据的方式。它是数据元素的集合，并且这些元素之间有特定的逻辑关系。常见的数据结构有线性数据结构和非线性数据结构两大类：

- **线性数据结构**：每个元素有且只有一个前驱和一个后继元素，通常表示为线性序列。典型的线性数据结构有：
  - **数组**：一组连续的存储空间，用于存储相同类型的数据元素，支持通过索引快速访问。
  - **链表**：由一系列节点组成，每个节点包含数据元素和指向下一个节点的指针，支持高效的插入和删除操作。
  - **栈**：遵循后进先出（LIFO）原则的线性结构。
  - **队列**：遵循先进先出（FIFO）原则的线性结构。

- **非线性数据结构**：数据元素之间不呈线性关系。典型的非线性数据结构有：
  - **树**：由节点和边组成的结构，节点有一个父节点和零个或多个子节点。常见的树结构有二叉树、平衡树、堆等。
  - **图**：由一组节点和连接节点的边组成，图中的边不一定具有方向，通常用于表示复杂关系。

数据结构的选择直接影响程序的执行效率和资源使用，因此需要根据具体应用的需求来选择合适的数据结构。

1.2.3 数据类型和抽象数据类型

- **数据类型**：是指一组具有相同性质的值的集合以及在这些值上定义的操作。数据类型可以分为**基本数据类型**和**用户定义数据类型**。
  - **基本数据类型**：如整型、浮点型、字符型、布尔型等，这些数据类型直接由编程语言提供。
  - **用户定义数据类型**：由程序员根据实际需求定义的数据类型，如结构体、类等。
- **抽象数据类型（ADT）**：抽象数据类型是数据结构的抽象表示，它定义了数据对象的操作接口，而不关心其具体实现。ADT关注的是如何进行数据操作，而非具体的实现细节。例如，栈、队列、列表等都可以看作抽象数据类型，它们通过一组操作（如入栈、出栈、插入、删除等）定义了数据的行为，但具体的存储方式可以是数组、链表或其他结构。

抽象数据类型的概念有助于程序设计的模块化和代码复用，通过抽象出数据的操作接口，可以让代码更具可维护性和扩展性。

1.3 抽象数据类型的表示与实现

抽象数据类型（Abstract Data Type, ADT）是数据结构的核心概念之一。它定义了一种数据模型以及在该模型上可以执行的操作，而不涉及具体实现的细节。具体而言，ADT包含数据和对数据的操作，而这些操作会封装在接口中。

抽象数据类型的表示

抽象数据类型的表示通常有两种方式：

1. **数学模型表示法**：
   - 通过集合和操作的数学定义来表示ADT。例如，栈（Stack）可以表示为一个具有特定操作的集合：
     - 操作：
       - `push(x)`：将元素x推入栈顶。
       - `pop()`：移除栈顶元素。
       - `top()`：返回栈顶元素但不移除它。
       - `isEmpty()`：检查栈是否为空。

   在数学模型中，我们主要关心数据的性质和操作，而不关注数据如何存储。
2. **程序实现表示法**：
   - 通过编程语言实现ADT。一般使用类（class）、接口（interface）等结构来封装数据和操作。例如，栈可以通过数组或链表来实现，但无论实现细节如何，栈的操作接口保持不变。这种方式关注的是如何实现这些操作的效率和细节。

抽象数据类型的实现

ADT的实现方式通常依赖于具体的数据结构。比如：

- 栈的实现
  ：可以通过数组、链表或者动态数组等结构来实现。
  - 使用**数组**时，栈的 `push` 和 `pop` 操作的时间复杂度为O(1)，但栈的容量固定，可能会出现溢出问题。
  - 使用**链表**时，栈可以动态扩展，避免了溢出问题，但是 `push` 和 `pop` 操作仍为O(1)。

通过接口和实现分离的方式，ADT的使用者可以依赖于抽象的操作，而不需要关心具体实现的细节。

1.4 算法和算法分析

在数据结构的研究中，算法的设计与分析同样重要。算法是用于解决问题的一组步骤，而算法分析则帮助我们理解和优化这些步骤的效率。

1.4.1 算法的定义及特性

- 算法的定义
  ：算法是为解决某个特定问题而设计的、有限的、明确的步骤集合。它应当具有以下特性：
  - **输入**：算法有零个或多个输入数据。
  - **输出**：算法有一个或多个输出结果。
  - **确定性**：算法的每一步都必须是明确的，不能存在歧义。
  - **有限性**：算法必须在有限的时间内终止，不得进入无限循环。
  - **可行性**：算法中的每个步骤都应该是可执行的。

一个好的算法不仅能够正确解决问题，还需要考虑它的执行效率，即时间复杂度和空间复杂度。

1.4.2 评价算法优劣的基本标准

评价算法优劣的标准主要包括以下几个方面：

- **时间复杂度**：衡量算法执行所需时间随输入规模增长的变化情况。通常用“大O符号”（如O(n)）来表示。
- **空间复杂度**：衡量算法执行过程中所需的内存空间随输入规模增长的变化情况。空间复杂度与时间复杂度密切相关，但关注的是内存使用。
- **可维护性**：一个好的算法应当易于理解和维护。代码应清晰、简洁，避免不必要的复杂性。
- **稳定性**：在处理输入数据中存在重复元素时，稳定算法能够保持相同元素的相对顺序。
- **适应性**：适应性强的算法能够处理多种不同规模和类型的输入数据。
- **可扩展性**：随着问题规模的增大，算法是否能够保持良好的性能表现。

1.4.3 算法的时间复杂度

时间复杂度是用来描述一个算法执行所需时间随输入规模（n）变化的规律。它是评估算法效率的一个重要指标。

- **常数时间复杂度** O(1)：表示算法的执行时间不随输入规模的增加而变化。例如，访问数组中的一个元素。
- **线性时间复杂度** O(n)：表示算法的执行时间与输入规模成正比。例如，遍历一个数组。
- **平方时间复杂度** $O(n^2)$：表示算法的执行时间与输入规模的平方成正比。例如，冒泡排序。
- **对数时间复杂度** O(log n)：表示算法的执行时间随着输入规模的增长呈对数增长。例如，二分查找。
- **线性对数时间复杂度** O(n log n)：常见于高效排序算法，如归并排序和快速排序。

时间复杂度不仅关心最坏情况，还常常需要分析平均时间复杂度，以确保算法在一般情况下的性能。

1.4.4 算法的空间复杂度

空间复杂度是用来描述一个算法在执行过程中所需的内存空间随着输入规模的变化而变化的规律。它与时间复杂度类似，但关注的是内存的消耗。

- **常数空间复杂度** O(1)：表示算法所需的内存空间不随输入规模的变化而变化。例如，使用固定大小的变量。
- **线性空间复杂度** O(n)：表示算法的内存空间需求与输入规模成正比。例如，创建一个大小为n的数组来存储输入数据。

在设计算法时，不仅要考虑时间复杂度，还需要考虑空间复杂度。在一些内存受限的环境中，空间复杂度可能比时间复杂度更重要。

第 2 章 线性表

线性表是数据结构中最基础和最常见的一种类型。它是一种数据元素的集合，满足“线性”结构的要求，即每个元素都与前一个和后一个元素具有确定的顺序关系。在线性表中，元素的插入、删除、查找操作通常依赖于元素的顺序或位置。

2.1 线性表的定义和特点

线性表的定义：线性表（Linear List）是由若干数据元素按线性顺序排列构成的集合。在这个集合中，每个元素（除第一个元素外）都有唯一的前驱元素，每个元素（除最后一个元素外）都有唯一的后继元素。

线性表的常见实现方式包括**顺序存储结构**（通常使用数组）和**链式存储结构**（通常使用链表）。

线性表的特点：

1. **顺序性**：线性表中的元素是有顺序的。每个元素都有明确的位置关系，一个元素仅有一个前驱和后继元素。
2. **插入与删除**：
   - 在顺序存储结构中，插入和删除操作可能需要移动大量元素，尤其是在数组的中间部分插入或删除时，操作复杂度较高。
   - 在链式存储结构中，插入和删除操作相对灵活，尤其是在表头或表尾进行操作时，复杂度通常为O(1)。
3. **访问性**：线性表支持按序号访问元素。对于顺序存储结构，访问元素的时间复杂度是O(1)，通过索引直接访问；而链式存储结构需要从头开始遍历，访问一个特定元素的时间复杂度是O(n)。
4. **结构灵活性**：线性表既可以是静态的（如数组），也可以是动态的（如链表）。顺序存储结构的大小在创建时固定，而链式存储结构可以根据需要动态扩展或收缩。
5. **支持顺序操作**：对线性表可以进行的基本操作包括：插入、删除、查找、修改、遍历等，这些操作大多是基于元素的顺序位置来执行。

2.2 案例引入

案例：学生成绩管理系统中的线性表

假设我们需要设计一个学生成绩管理系统，系统中需要记录学生的成绩信息，并能进行如下操作：

- 查询某个学生的成绩。
- 按照成绩排序（升序或降序）。
- 插入新的成绩数据。
- 删除某个学生的成绩。

在这个系统中，我们可以将每个学生的成绩表示为线性表。每个元素包含学生的姓名和成绩信息。线性表可以帮助我们高效地进行查询、插入和删除操作。

1. **查询操作**：使用顺序存储结构（如数组），可以根据学生的索引位置快速查询成绩。若采用链表结构，也可以进行查找，但需要遍历链表来找到对应节点。
2. **排序操作**：如果学生成绩存储在一个线性表中，我们可以使用排序算法（如冒泡排序、快速排序等）对其进行排序。
3. **插入和删除操作**：对于顺序存储结构，插入和删除操作可能需要移动元素，而对于链表，只需要改变指针即可完成插入和删除，效率较高。

这个案例帮助我们理解线性表在实际应用中的作用，并让我们能根据需求选择合适的存储结构。

2.3 线性表的类型定义

线性表有多种类型，常见的类型包括：**顺序表**（Array List）和**链表**（Linked List）。这两种类型的线性表在内存布局和操作效率上有所不同。

1. **顺序表**：顺序表是线性表的一种实现方式，它使用一段连续的内存空间来存储元素。在顺序表中，每个元素通过下标进行访问，具有高效的随机访问能力。
   - 优点：
     - 支持O(1)时间复杂度的访问操作，能够通过索引快速访问元素。
     - 空间利用率高（在不发生动态扩展的情况下），适合存储大小已知且变化不大的数据。
   - 缺点：
     - 插入和删除操作的时间复杂度为O(n)，因为在数组中间插入或删除元素时，需要移动其他元素。
     - 如果数组的大小固定，则无法动态增长或缩小，可能会浪费空间或造成内存溢出。
2. **链表**：链表是一种通过指针将多个数据元素连接起来的线性结构。链表的每个元素包含两个部分：数据域（存储数据）和指针域（指向下一个元素）。链表的长度可以动态变化，因此适合用于需要频繁插入和删除操作的场景。
   - 优点：
     - 插入和删除操作非常高效，尤其是在表头或表尾进行操作时，时间复杂度为O(1)。
     - 动态内存分配，适用于数据量不固定或变化频繁的情况。
   - 缺点：
     - 访问元素的时间复杂度为O(n)，因为需要从头结点开始遍历，无法实现快速的随机访问。
     - 每个元素都需要额外的存储空间来存储指针，因此比顺序表浪费更多的内存。
3. **其他类型的线性表**：除了顺序表和链表，还有一些其他类型的线性表，例如：
   - **静态链表**：将链表的指针域存储在一个数组中，解决了动态内存分配的问题，但插入和删除操作仍然保持O(1)。
   - **循环链表**：链表的最后一个元素的指针指向第一个元素，形成一个闭环，适用于某些需要循环访问的场景。

2.4 线性表的顺序表示和实现

线性表的顺序表示是一种基于数组的实现方式，它通过一段连续的内存空间来存储线性表中的元素。顺序表中的元素按顺序排列，每个元素可以通过索引直接访问。顺序表示的优点是具有快速的随机访问能力，但也存在一些限制，如插入和删除操作的效率较低。

2.4.1 线性表的顺序表示

在顺序表示中，线性表中的每个元素依次存储在一个数组中，数组的索引就是元素的位置。假设我们有一个长度为n的线性表，它的顺序表示可以通过一个大小为n的数组来实现：

```
int arr[n]; // 数组arr用于存储n个元素
```

每个数组元素对应线性表中的一个数据元素。顺序表示的线性表有以下特点：

- **支持随机访问**：通过下标可以直接访问任意元素，访问时间为O(1)。
- **连续存储**：数组中元素的存储位置是连续的，这使得顺序表的内存布局非常紧凑。
- **固定大小**：顺序表在创建时必须确定大小，一旦创建，大小不可改变（除非使用动态数组）。
- **插入和删除效率低**：在数组的中间插入或删除元素时，需要移动大量元素，时间复杂度为O(n)。

顺序表示的线性表的基本结构通常包含以下两个要素：

1. **数组**：用于存储线性表的元素。
2. **长度**：记录线性表当前的元素个数。

例如，定义一个存储整数的顺序表结构：

```
class SeqList {
private:
    int* arr;  // 动态数组
    int length; // 当前元素个数
    int capacity; // 数组容量

public:
    SeqList(int size) {
        arr = new int[size];
        length = 0;
        capacity = size;
    }

    ~SeqList() {
        delete[] arr;
    }

    // 其他操作...
};
```


2.4.2 顺序表中基本操作的实现

顺序表中常见的基本操作包括：插入元素、删除元素、查找元素、修改元素、遍历等。下面我们将通过C++代码展示这些基本操作的实现。

1. 插入操作

插入操作是将一个新元素插入到顺序表中的某个位置。插入时需要考虑以下几种情况：

- 如果插入位置在表尾，只需要直接将元素插入。
- 如果插入位置在表中间，需要移动元素以腾出空间。

```
// 插入操作
void insert(int position, int value) {
    if (position < 0 || position > length) {
        cout << "Invalid position!" << endl;
        return;
    }

    if (length == capacity) {
        cout << "Sequence List is full!" << endl;
        return;
    }

    // 从后往前移动元素
    for (int i = length - 1; i >= position; --i) {
        arr[i + 1] = arr[i];
    }

    arr[position] = value; // 插入新元素
    length++; // 更新长度
}
```

- **时间复杂度**：O(n)，最坏情况下需要移动所有元素。

2. 删除操作

删除操作是将顺序表中指定位置的元素删除。删除时需要考虑以下几种情况：

- 删除元素后，表中其后的元素需要向前移动，填补空缺位置。

```
// 删除操作
void remove(int position) {
    if (position < 0 || position >= length) {
        cout << "Invalid position!" << endl;
        return;
    }

    // 从当前位置开始，元素依次向前移动
    for (int i = position; i < length - 1; ++i) {
        arr[i] = arr[i + 1];
    }

    length--; // 更新长度
}
```

- **时间复杂度**：O(n)，最坏情况下需要移动所有元素。

3. 查找操作

查找操作是根据值或者位置找到顺序表中的元素。根据需求可以是按值查找或按位置查找。

```
// 按位置查找
int get(int position) {
    if (position < 0 || position >= length) {
        cout << "Invalid position!" << endl;
        return -1;
    }
    return arr[position];
}

// 按值查找
int find(int value) {
    for (int i = 0; i < length; ++i) {
        if (arr[i] == value) {
            return i; // 返回值所在位置
        }
    }
    return -1; // 未找到
}
```

- **时间复杂度**：查找操作的时间复杂度为O(n)，最坏情况下需要遍历整个数组。

4. 修改操作

修改操作是根据位置修改顺序表中指定位置的元素。

```
// 修改操作
void modify(int position, int value) {
    if (position < 0 || position >= length) {
        cout << "Invalid position!" << endl;
        return;
    }

    arr[position] = value;
}
```

- **时间复杂度**：O(1)，只需要直接访问并修改指定位置的元素。

5. 遍历操作

遍历操作是依次访问顺序表中的每个元素，常用于打印输出。

```
// 遍历操作
void print() {
    for (int i = 0; i < length; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

- **时间复杂度**：O(n)，需要遍历整个数组。

完整代码示例

以下是顺序表的完整实现，包括插入、删除、查找、修改和遍历操作：

```
#include <iostream>
using namespace std;

class SeqList {
private:
    int* arr;  // 动态数组
    int length; // 当前元素个数
    int capacity; // 数组容量

public:
    // 构造函数
    SeqList(int size) {
        arr = new int[size];
        length = 0;
        capacity = size;
    }

    // 析构函数
    ~SeqList() {
        delete[] arr;
    }

    // 插入操作
    void insert(int position, int value) {
        if (position < 0 || position > length) {
            cout << "Invalid position!" << endl;
            return;
        }

        if (length == capacity) {
            cout << "Sequence List is full!" << endl;
            return;
        }

        for (int i = length - 1; i >= position; --i) {
            arr[i + 1] = arr[i];
        }

        arr[position] = value;
        length++;
    }

    // 删除操作
    void remove(int position) {
        if (position < 0 || position >= length) {
            cout << "Invalid position!" << endl;
            return;
```

```
        }

        for (int i = position; i < length - 1; ++i) {
            arr[i] = arr[i + 1];
        }

        length--;
    }

    // 查找操作
    int get(int position) {
        if (position < 0 || position >= length) {
            cout << "Invalid position!" << endl;
            return -1;
        }
        return arr[position];
    }

    int find(int value) {
        for (int i = 0; i < length; ++i) {
            if (arr[i] == value) {
                return i;
            }
        }
        return -1;
    }

    // 修改操作
    void modify(int position, int value) {
        if (position < 0 || position >= length) {
            cout << "Invalid position!" << endl;
            return;
        }

        arr[position] = value;
    }

    // 遍历操作
    void print() {
        for (int i = 0; i < length; ++i) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    SeqList list(10);

    // 插入操作
    list.insert(0, 10);
    list.insert(1, 20);

    list.insert(2, 30);
```

```
    list.print(); // 输出: 10 20 30

    // 删除操作
    list.remove(1);
    list.print(); // 输出: 10 30

    // 查找操作
    cout << "Index of 30: " << list.find(30) << endl; // 输出: 1

    // 修改操作
    list.modify(0, 50);
    list.print(); // 输出: 50 30

    return 0;
}
```

总结

- **顺序表的优点**：支持快速的随机访问，结构简单。
- **顺序表的缺点**：插入和删除操作的效率较低，尤其是在中间位置进行操作时需要移动大量元素。
- **适用场景**：当数据量不大且插入和删除

2.4 线性表的顺序表示和实现

线性表的顺序表示是指使用一段连续的内存空间来存储线性表的元素。顺序表示方法使得线性表的每个元素都可以通过下标进行直接访问，能够高效地进行查找和修改操作。然而，插入和删除操作可能需要移动元素，尤其是在中间位置插入或删除时，性能较差。

2.4.1 线性表的顺序表示

顺序表（Sequence List）是线性表的顺序存储结构，其基本思想是用一段连续的内存空间来存储元素，通常使用数组来实现。顺序表的每个元素通过索引（下标）进行访问，支持高效的随机访问。

顺序表的优缺点：

- 优点
  :
  - 支持O(1)时间复杂度的随机访问，能够快速定位和操作元素。
  - 内存连续存储，空间利用率高，访问速度快。
- 缺点
  :
  - 插入和删除操作需要移动元素，尤其是在表的中间进行插入或删除时，时间复杂度为O(n)。
  - 如果表的大小固定，且元素超过最大容量时，会导致溢出，需要重新分配内存空间。

顺序表的结构如下图所示：

```
| Element 1 | Element 2 | Element 3 | ... | Element n |
```

每个元素的位置通过索引来确定，索引从0到n-1。

顺序表的常见操作：

- **插入**：在指定位置插入元素，通常需要移动插入位置之后的元素。
- **删除**：删除指定位置的元素，通常需要移动删除位置之后的元素。
- **查找**：通过索引直接访问指定位置的元素。
- **修改**：通过索引修改指定位置的元素。

2.4.2 顺序表中基本操作的实现

在顺序表中，常见的基本操作包括：初始化、插入、删除、查找、修改、遍历等。下面是这些操作的C++代码实现，并附有详细注释。

1. 顺序表的结构定义

```
#include <iostream>
using namespace std;

// 定义顺序表类
class SeqList {
private:
    int* arr;           // 动态数组，存储线性表的元素
    int capacity;       // 顺序表的最大容量
    int length;         // 当前顺序表中元素的个数

public:
    // 构造函数：初始化顺序表
    SeqList(int cap) {
        capacity = cap;
        length = 0;
        arr = new int[capacity]; // 动态分配内存
    }

    // 析构函数：释放动态分配的内存
    ~SeqList() {
        delete[] arr;
    }

    // 打印顺序表中的所有元素
    void print() {
        for (int i = 0; i < length; i++) {
            cout << arr[i] << " ";
        }
        cout << endl;
    }

    // 插入元素，时间复杂度：O(n)
    bool insert(int index, int value) {
        // 检查索引是否合法
        if (index < 0 || index > length || length == capacity) {
            cout << "Invalid position or sequence full!" << endl;
            return false;
        }
```

```
    // 从末尾开始移动元素
    for (int i = length; i > index; i--) {
        arr[i] = arr[i - 1];
    }

    arr[index] = value;  // 插入新元素
    length++;  // 增加元素个数
    return true;
}

// 删除元素，时间复杂度：O(n)
bool remove(int index) {
    // 检查索引是否合法
    if (index < 0 || index >= length) {
        cout << "Invalid position!" << endl;
        return false;
    }

    // 从删除位置后一个元素开始移动
    for (int i = index; i < length - 1; i++) {
        arr[i] = arr[i + 1];
    }

    length--;  // 减少元素个数
    return true;
}

// 查找元素的索引，时间复杂度：O(n)
int find(int value) {
    for (int i = 0; i < length; i++) {
        if (arr[i] == value) {
            return i;  // 找到元素，返回索引
        }
    }
    return -1;  // 未找到，返回-1
}

// 修改指定位置的元素
bool modify(int index, int value) {
    // 检查索引是否合法
    if (index < 0 || index >= length) {
        cout << "Invalid position!" << endl;
        return false;
    }

    arr[index] = value;  // 修改元素
    return true;
}

// 获取顺序表的长度
int getLength() {
    return length;

}
```

```
};

// 测试顺序表操作
int main() {
    SeqList list(10);  // 创建一个容量为10的顺序表

    list.insert(0, 5);  // 插入元素5
    list.insert(1, 10); // 插入元素10
    list.insert(2, 15); // 插入元素15

    cout << "Sequence List after insertions: ";
    list.print();  // 输出: 5 10 15

    list.remove(1);  // 删除位置为1的元素（元素10）

    cout << "Sequence List after removal: ";
    list.print();  // 输出: 5 15

    list.modify(1, 20);  // 修改位置1的元素为20

    cout << "Sequence List after modification: ";
    list.print();  // 输出: 5 20

    int index = list.find(5);  // 查找元素5的位置
    if (index != -1) {
        cout << "Element 5 found at index: " << index << endl;  // 输出: Element 5 found at
index: 0
    }

    return 0;
}
```


代码说明:

1. **SeqList类**:
   - `arr`: 动态数组，用于存储顺序表的元素。
   - `capacity`: 顺序表的最大容量。
   - `length`: 顺序表中当前元素的个数。
   - 构造函数和析构函数分别用于初始化和释放内存。
2. **insert()**:
   - 插入操作，时间复杂度为O(n)。首先检查插入位置是否合法，然后从末尾开始将元素后移，最后插入新元素。
3. **remove()**:
   - 删除操作，时间复杂度为O(n)。首先检查删除位置是否合法，然后将删除位置后面的元素前移。
4. **find()**:
   - 查找操作，时间复杂度为O(n)。通过遍历顺序表，查找元素的位置，找到返回索引，找不到返回-1。
5. **modify()**:
   - 修改操作，时间复杂度为O(1)。根据给定的索引直接修改对应位置的元素。

6. **print()**：
    - 输出顺序表的所有元素。

总结

顺序表示的线性表通过数组实现，适合快速访问和修改操作，但在插入和删除元素时，尤其是在表的中间进行操作时，性能较差。通过这些基本操作的实现，我们可以理解顺序表的核心思想，并在实际应用中根据需求选择合适的存储结构。

2.5 线性表的链式表示和实现

链式表示是线性表的另一种实现方式，它通过指针将元素连接起来，而不是在内存中使用连续的空间。链式表示可以支持动态变化的长度，因此在插入和删除操作时相较于顺序表更加灵活。

链式表示包括单链表、双向链表和循环链表。下面我们将逐一介绍这些链表的定义、操作以及实现。

2.5.1 单链表的定义和表示

单链表（Singly Linked List）是由若干个节点组成的线性表，每个节点包含两部分：

1. **数据域**：存储数据。
2. **指针域**：存储指向下一个节点的指针。

单链表的特点是每个节点仅指向后继节点，因此只能单向访问。

单链表的结构：

```
struct Node {
    int data;        // 数据域，存储元素值
    Node* next;      // 指针域，指向下一个节点
};
```

单链表的基本操作：

- **插入**：在指定位置插入节点。
- **删除**：删除指定位置的节点。
- **查找**：查找指定值的节点。
- **修改**：修改指定位置的节点数据。
- **遍历**：从头到尾访问所有节点。

2.5.2 单链表基本操作的实现

下面是实现单链表基本操作的C++代码示例，带有详细注释。

```
#include <iostream>
using namespace std;

// 定义单链表节点结构
struct Node {
    int data;       // 数据域
    Node* next;     // 指向下一个节点的指针

};
```

```
// 单链表类
class SinglyLinkedList {
private:
    Node* head;  // 链表的头指针

public:
    // 构造函数：初始化链表
    SinglyLinkedList() {
        head = nullptr;  // 初始化时头指针为空
    }

    // 析构函数：释放链表的所有节点
    ~SinglyLinkedList() {
        Node* temp;
        while (head != nullptr) {
            temp = head;
            head = head->next;  // 移动头指针
            delete temp;        // 删除节点
        }
    }

    // 向链表尾部添加元素
    void append(int value) {
        Node* newNode = new Node{value, nullptr};  // 创建新节点
        if (head == nullptr) {
            head = newNode;  // 如果链表为空，新的节点作为头节点
        } else {
            Node* temp = head;
            while (temp->next != nullptr) {
                temp = temp->next;  // 找到链表的尾节点
            }
            temp->next = newNode;  // 将新节点连接到尾节点
        }
    }

    // 在指定位置插入元素
    void insertAt(int index, int value) {
        Node* newNode = new Node{value, nullptr};
        if (index == 0) {  // 插入到头部
            newNode->next = head;
            head = newNode;
        } else {
            Node* temp = head;
            for (int i = 0; i < index - 1 && temp != nullptr; i++) {
                temp = temp->next;  // 遍历到指定位置
            }
            if (temp != nullptr) {
                newNode->next = temp->next;  // 将新节点的next指向目标节点
                temp->next = newNode;        // 将前一个节点的next指向新节点
            } else {
                cout << "Invalid position!" << endl;

                delete newNode;
```

```
        }
    }
}

// 删除指定位置的节点
void deleteAt(int index) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    Node* temp;
    if (index == 0) {  // 删除头节点
        temp = head;
        head = head->next;
        delete temp;
    } else {
        Node* prev = head;
        for (int i = 0; i < index - 1 && prev->next != nullptr; i++) {
            prev = prev->next;
        }
        if (prev->next != nullptr) {
            temp = prev->next;
            prev->next = temp->next;  // 删除节点
            delete temp;
        } else {
            cout << "Invalid position!" << endl;
        }
    }
}

// 查找指定值的节点并返回位置
int find(int value) {
    Node* temp = head;
    int index = 0;
    while (temp != nullptr) {
        if (temp->data == value) {
            return index;  // 找到返回索引
        }
        temp = temp->next;
        index++;
    }
    return -1;  // 未找到返回-1
}

// 打印链表元素
void print() {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }

    cout << endl;
```

```
    }
};

// 测试单链表的操作
int main() {
    SinglyLinkedList list;

    // 向链表添加元素
    list.append(10);
    list.append(20);
    list.append(30);
    cout << "List after append: ";
    list.print();  // 输出: 10 20 30

    // 在指定位置插入元素
    list.insertAt(1, 15);
    cout << "List after insert at position 1: ";
    list.print();  // 输出: 10 15 20 30

    // 删除指定位置的元素
    list.deleteAt(2);
    cout << "List after delete at position 2: ";
    list.print();  // 输出: 10 15 30

    // 查找元素位置
    int index = list.find(15);
    if (index != -1) {
        cout << "Element 15 found at index: " << index << endl;  // 输出: Element 15 found at
index: 1
    }

    return 0;
}
```


代码说明:

1. **Node结构**：表示链表的节点，每个节点包含一个数据域和一个指向下一个节点的指针。
2. SinglyLinkedList类
   :
   - **append()**：向链表尾部添加元素。
   - **insertAt()**：在指定位置插入元素。如果位置为0，则插入到头部，否则遍历到指定位置插入。
   - **deleteAt()**：删除指定位置的节点。如果位置为0，删除头节点，否则找到前一个节点并删除。
   - **find()**：查找指定值的节点并返回索引。
   - **print()**：打印链表中所有节点的值。

2.5.3 循环链表

循环链表是指链表的最后一个节点指向头节点，形成一个闭环。循环链表有两种形式：**单向循环链表**和**双向循环链表**。单向循环链表的每个节点只有一个指向下一个节点的指针。

循环链表的优点：它的尾部指针可以直接连接到头部，因此在尾部插入和删除操作时更加高效。

```
// 定义循环链表节点结构
struct Node {
    int data;
    Node* next;
};

// 定义循环链表类
class CircularLinkedList {
private:
    Node* head;

public:
    CircularLinkedList() {
        head = nullptr;
    }

    ~CircularLinkedList() {
        if (head != nullptr) {
            Node* temp = head;
            do {
                Node* next = temp->next;
                delete temp;
                temp = next;
            } while (temp != head);
        }
    }

    // 向循环链表中添加节点
    void append(int value) {
        Node* newNode = new Node{value, nullptr};
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;  // 循环链表的尾节点指向头节点
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;  // 新节点的next指向头节点
        }
    }

    // 打印循环链表的所有元素
    void print() {
        if (head == nullptr) return;
        Node* temp = head;
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        cout << endl;

    }
```

```
};

// 测试循环链表
int main() {
    CircularLinkedList clist;
    clist.append(10);
    clist.append(20);
    clist.append(30);
    cout << "Circular Linked List: ";
    clist.print();  // 输出: 10 20
```

30

```
return 0;
```

}

```
#### 2.5.4 双向链表

**双向链表** (Doubly Linked List) 是每个节点除了指向下一个节点的指针外，还指向前一个节点。双向链表支持从两端进行操作，因此可以更加高效地进行插入和删除。

```cpp
// 定义双向链表节点结构
struct Node {
    int data;
    Node* next;
    Node* prev;  // 指向前一个节点
};

// 定义双向链表类
class DoublyLinkedList {
private:
    Node* head;
    Node* tail;

public:
    DoublyLinkedList() {
        head = tail = nullptr;
    }

    ~DoublyLinkedList() {
        Node* temp = head;
        while (temp != nullptr) {
            Node* next = temp->next;
            delete temp;
            temp = next;
        }
    }


    // 向双向链表尾部添加节点
```
```

```
    void append(int value) {
        Node* newNode = new Node{value, nullptr, nullptr};
        if (head == nullptr) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    // 打印双向链表的所有元素
    void print() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

// 测试双向链表
int main() {
    DoublyLinkedList dlist;
    dlist.append(10);
    dlist.append(20);
    dlist.append(30);
    cout << "Doubly Linked List: ";
    dlist.print();  // 输出: 10 20 30

    return 0;
}
```


总结

- **单链表**：每个节点只包含一个指向下一个节点的指针，插入和删除操作在链表中间时非常高效。
- **循环链表**：单向链表的尾节点指向头节点，形成一个闭环，适合需要循环遍历的应用场景。
- **双向链表**：每个节点包含指向前后节点的指针，支持从两端进行操作，插入和删除操作更为灵活。

2.6 顺序表和链表的比较

在数据结构的设计中，**顺序表**（顺序存储的线性表）和**链表**（链式存储的线性表）各有优缺点，它们在**空间性能**和**时间性能**上的表现有所不同。下面我们将通过这两个方面进行详细的比较。

2.6.1 空间性能的比较

- **顺序表**：
  - 优点
    :
    - 空间上是连续的，可以通过数组的下标快速定位元素。

        - 内存使用较为紧凑，适合较为静态的数据集合。
    - 缺点
      :
        - 需要预先分配一定大小的内存，容量固定，扩展困难。
        - 如果实际存储的数据比预设的空间少，可能会造成内存浪费；如果数据量大于预设容量，则需要重新分配内存。
        - 由于需要连续的内存空间，可能会出现内存分配失败的情况（尤其在数据量很大时）。
- **链表**:
    - 优点
      :
        - 动态分配内存，链表节点的内存分配是独立的，可以根据实际需要增减节点。
        - 不需要连续的内存空间，适合不确定大小的集合。
    - 缺点
      :
        - 每个节点除了数据外，还需要额外的空间来存储指针（对于单链表是指向下一个节点的指针，对于双向链表是指向前后两个节点的指针）。
        - 总体内存使用比顺序表要多，因为每个节点都包含了指针。

总结:

- 顺序表的空间性能较优，尤其在已知数据量固定时，能够充分利用内存。
- 链表的空间利用率较低，因为每个元素都需要额外存储指针，但是它能够动态扩展，适合数据量不确定的情况。

2.6.2 时间性能的比较

在时间性能方面，主要考虑以下几个操作：查找、插入、删除等。

1. **查找操作**:
    - 顺序表
      :
        - 查找操作是通过数组下标进行的，时间复杂度为 **O(1)**。
        - 对于顺序查找（无序的情况），时间复杂度为 **O(n)**。
    - 链表
      :
        - 对于链表，查找操作需要从头节点开始遍历，直到找到目标元素。最坏情况下时间复杂度为 **O(n)**，即需要遍历整个链表。
        - 对于有序链表，可以通过顺序查找或者跳跃查找，但一般依然是 **O(n)**。

    总结:
    - 顺序表的查找性能较优，能够直接通过下标访问元素。
    - 链表的查找性能较差，需要逐个节点进行遍历。
2. **插入操作**:
    - 顺序表

：

- 如果是在末尾插入元素，时间复杂度为 **O(1)**。
- 如果是插入到中间或前面，则需要将插入位置后的所有元素后移，时间复杂度为 **O(n)**。

- 链表

：

- 在链表中插入元素，时间复杂度为 **O(1)**，前提是给定插入位置的指针（例如，在头部或尾部插入）。
- 如果没有给定位置的指针，仍然需要遍历链表定位插入位置，时间复杂度为 **O(n)**。

总结：

- 链表的插入操作在已知位置时较为高效，尤其是在表头或表尾插入时。
- 顺序表的插入操作需要移动元素，性能较差，尤其是在中间插入时。

3. **删除操作**：
   - 顺序表

：

- 删除操作类似于插入操作。如果是删除末尾元素，时间复杂度为 **O(1)**。
- 如果是删除中间元素，则需要将删除位置后的所有元素前移，时间复杂度为 **O(n)**。

- 链表

：

- 删除操作需要找到目标节点，并调整指针，时间复杂度为 **O(1)**，前提是已知目标节点。
- 如果需要先遍历链表找到目标节点，时间复杂度为 **O(n)**。

总结：

- 链表的删除操作在已知位置时也非常高效，而顺序表在删除操作时，如果涉及到中间元素，性能较差。

C++ 代码实现：顺序表与链表的性能比较

为了更好地理解空间和时间性能的差异，以下是一个简单的C++实现，用于比较顺序表和链表在插入、删除和查找操作上的性能。

```
#include <iostream>
#include <ctime>  // 用于计算时间
#include <vector>
using namespace std;

// 定义顺序表类
class SeqList {
private:
    vector<int> arr;  // 使用vector动态分配内存

public:
    // 插入操作
    void insert(int value) {
        arr.push_back(value);  // 向尾部插入元素
    }


    // 查找操作
```

```
    int find(int value) {
        for (int i = 0; i < arr.size(); i++) {
            if (arr[i] == value) {
                return i;
            }
        }
        return -1;
    }

    // 删除操作
    void remove(int index) {
        if (index >= 0 && index < arr.size()) {
            arr.erase(arr.begin() + index);  // 删除指定位置的元素
        }
    }

    // 获取表的大小
    int size() {
        return arr.size();
    }
};

// 定义链表节点
struct Node {
    int data;
    Node* next;
};

// 定义链表类
class LinkedList {
private:
    Node* head;

public:
    LinkedList() {
        head = nullptr;
    }

    // 插入操作
    void insert(int value) {
        Node* newNode = new Node{value, head};
        head = newNode;  // 在头部插入
    }

    // 查找操作
    int find(int value) {
        Node* temp = head;
        int index = 0;
        while (temp != nullptr) {
            if (temp->data == value) {
                return index;
            }

            temp = temp->next;
```

```
            index++;
        }
        return -1;
    }

    // 删除操作
    void remove(int value) {
        Node* temp = head;
        Node* prev = nullptr;
        while (temp != nullptr) {
            if (temp->data == value) {
                if (prev == nullptr) {
                    head = temp->next;
                } else {
                    prev->next = temp->next;
                }
                delete temp;
                return;
            }
            prev = temp;
            temp = temp->next;
        }
    }

    // 获取链表长度
    int size() {
        int count = 0;
        Node* temp = head;
        while (temp != nullptr) {
            count++;
            temp = temp->next;
        }
        return count;
    }
};

// 测试函数
void testPerformance() {
    const int N = 100000;  // 测试数据规模

    // 测试顺序表性能
    SeqList seqList;
    clock_t start = clock();
    for (int i = 0; i < N; i++) {
        seqList.insert(i);  // 向顺序表插入元素
    }
    for (int i = 0; i < N; i++) {
        seqList.find(i);  // 查找顺序表中的元素
    }
    for (int i = 0; i < N; i++) {
        seqList.remove(0);  // 删除顺序表中的元素
    }

    clock_t end = clock();
```

```
    cout << "SeqList time: " << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;

    // 测试链表性能
    LinkedList linkedList;
    start = clock();
    for (int i = 0; i < N; i++) {
        linkedList.insert(i);  // 向链表插入元素
    }
    for (int i = 0; i < N; i++) {
        linkedList.find(i);  // 查找链表中的元素
    }
    for (int i = 0; i < N; i++) {
        linkedList.remove(i);  // 删除链表中的元素
    }
    end = clock();
    cout << "LinkedList time: " << double(end - start) / CLOCKS_PER_SEC << " seconds" << endl;
}

int main() {
    testPerformance();
    return 0;
}
```

代码说明：

1. **顺序表的实现**：使用 `vector` 来模拟顺序表。提供插入、查找和删除操作。
2. **链表的实现**：使用结构体 `Node` 来表示链表的节点，

提供插入、查找和删除操作。3. **性能测试**：通过插入、查找和删除操作来测试顺序表和链表在大规模数据（`N = 100000`）下的性能表现。通过 `clock_t` 测量时间。

输出：

代码运行后将显示顺序表和链表的操作时间，帮助我们了解它们在大数据量下的性能差异。

总结

- **顺序表**：由于内存是连续分配的，因此在查找操作时速度较快，但在插入和删除操作时需要移动元素，特别是在中间插入和删除时效率较低。
- **链表**：动态内存分配，不需要连续空间，适合经常插入和删除的场景，但查找操作较慢，因为需要逐个遍历节点。

2.7 线性表的应用

线性表作为一种常用的数据结构，在计算机科学和工程中有着广泛的应用。它不仅可以用于表示简单的线性数据集合，还可以被用来实现许多算法和解决实际问题。在线性表的应用中，**合并操作**是一个非常常见且重要的操作，尤其是在处理排序、集合运算等问题时。

2.7.1 线性表的合并

线性表的合并指的是将两个或多个线性表合并成一个线性表，合并后的新表包含所有原表的元素。在进行合并时，通常有两种方式：

1. **简单合并**：直接将一个线性表的元素添加到另一个线性表的末尾。
2. **合并时去重**：合并的同时去掉重复的元素，确保结果中每个元素的唯一性。

简单合并

简单合并线性表可以通过遍历两个表，将一个表的所有元素逐个插入到另一个表中。此操作的时间复杂度通常为**O(n)**，其中 n 是两个表中元素的总数。

C++代码实现：

```
#include <iostream>
#include <vector>
using namespace std;

// 合并两个线性表
void mergeLists(vector<int>& list1, vector<int>& list2) {
    // 将list2的元素添加到list1中
    for (int i = 0; i < list2.size(); i++) {
        list1.push_back(list2[i]);
    }
}

// 打印线性表
void printList(const vector<int>& list) {
    for (int i = 0; i < list.size(); i++) {
        cout << list[i] << " ";
    }
    cout << endl;
}

int main() {
    // 初始化两个线性表
    vector<int> list1 = {1, 2, 3, 4};
    vector<int> list2 = {5, 6, 7, 8};

    // 打印原始线性表
    cout << "List1: ";
    printList(list1);
    cout << "List2: ";
    printList(list2);

    // 合并两个线性表
    mergeLists(list1, list2);

    // 打印合并后的线性表
    cout << "Merged List: ";
    printList(list1);

    return 0;
}
```


代码说明：

- `mergeLists` 函数用于合并两个线性表，将 `list2` 的元素依次加入到 `list1` 中。
- `printList` 用于打印线性表的元素。

输出示例：

```
List1: 1 2 3 4
List2: 5 6 7 8
Merged List: 1 2 3 4 5 6 7 8
```

合并时去重

当合并两个线性表时，如果我们要求合并后的结果中没有重复的元素，则需要在合并时去除重复元素。我们可以使用哈希表或排序后去重的方式来实现。

C++代码实现（合并并去重）：

```
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;

// 合并两个线性表并去重
void mergeAndRemoveDuplicates(vector<int>& list1, vector<int>& list2) {
    unordered_set<int> uniqueSet;

    // 将list1的元素插入到集合中，自动去重
    for (int i = 0; i < list1.size(); i++) {
        uniqueSet.insert(list1[i]);
    }

    // 将list2的元素插入到集合中，自动去重
    for (int i = 0; i < list2.size(); i++) {
        uniqueSet.insert(list2[i]);
    }

    // 清空list1，并将去重后的元素按顺序添加回list1
    list1.clear();
    for (const auto& elem : uniqueSet) {
        list1.push_back(elem);
    }
}

// 打印线性表
void printList(const vector<int>& list) {
    for (int i = 0; i < list.size(); i++) {
        cout << list[i] << " ";
    }
    cout << endl;
}

int main() {

    // 初始化两个线性表
```

```
    vector<int> list1 = {1, 2, 3, 4};
    vector<int> list2 = {3, 4, 5, 6};

    // 打印原始线性表
    cout << "List1: ";
    printList(list1);
    cout << "List2: ";
    printList(list2);

    // 合并并去重
    mergeAndRemoveDuplicates(list1, list2);

    // 打印合并后的线性表
    cout << "Merged and Unique List: ";
    printList(list1);

    return 0;
}
```

代码说明：

- `mergeAndRemoveDuplicates` 函数利用 `unordered_set` 来去重，因为 `unordered_set` 在插入元素时会自动排除重复的元素。
- 合并后的结果将元素从 `unordered_set` 中提取，并按顺序放回 `list1` 中。

输出示例：

```
List1: 1 2 3 4
List2: 3 4 5 6
Merged and Unique List: 1 2 3 4 5 6
```


2.7.2 有序表的合并

如果合并的两个线性表已经是**有序的**，我们可以通过一种更高效的方式来合并它们，而不需要像简单合并那样依次插入每个元素。对于两个有序的线性表，我们可以采用**双指针法**（类似于归并排序中的合并操作）来进行合并。这个方法的时间复杂度为 **O(n)**，其中 n 是两个表中元素的总数。

C++代码实现（有序表合并）：

```
#include <iostream>
#include <vector>
using namespace std;

// 有序表合并
void mergeSortedLists(const vector<int>& list1, const vector<int>& list2, vector<int>& result) {
    int i = 0, j = 0;

    // 合并过程
    while (i < list1.size() && j < list2.size()) {
        if (list1[i] < list2[j]) {
            result.push_back(list1[i]);
            i++;
```

```
        } else {
            result.push_back(list2[j]);
            j++;
        }
    }

    // 将剩余的元素添加到结果中
    while (i < list1.size()) {
        result.push_back(list1[i]);
        i++;
    }

    while (j < list2.size()) {
        result.push_back(list2[j]);
        j++;
    }
}

// 打印线性表
void printList(const vector<int>& list) {
    for (int i = 0; i < list.size(); i++) {
        cout << list[i] << " ";
    }
    cout << endl;
}

int main() {
    // 初始化两个有序表
    vector<int> list1 = {1, 3, 5, 7};
    vector<int> list2 = {2, 4, 6, 8};
    vector<int> result;

    // 打印原始有序表
    cout << "List1: ";
    printList(list1);
    cout << "List2: ";
    printList(list2);

    // 合并有序表
    mergeSortedLists(list1, list2, result);

    // 打印合并后的有序表
    cout << "Merged Sorted List: ";
    printList(result);

    return 0;
}
```


代码说明：

- `mergeSortedLists` 函数使用两个指针 `i` 和 `j` 分别指向两个有序线性表的当前元素。
- 比较两个元素的大小，将较小的元素加入到结果表 `result` 中，然后移动对应的指针。
- 最后，将未遍历完的表中的剩余元素加入到结果表中。

输出示例：

```
List1: 1 3 5 7
List2: 2 4 6 8
Merged Sorted List: 1 2 3 4 5 6 7 8
```

总结

- **线性表的合并**：简单合并和去重合并是合并操作的常见应用。简单合并直接将一个表的元素插入到另一个表中；去重合并可以通过哈希表或排序后去重来实现。
- **有序表的合并**：对于已排序的表，使用双指针法进行合并，时间复杂度为 **O(n)**，比直接合并更高效。

第 3 章 栈和队列

栈（Stack）和队列（Queue）是两种常见的数据结构，它们都属于线性表结构，但在操作方式上有所不同。栈遵循的是“后进先出”（LIFO，Last In First Out）的原则，而队列遵循的是“先进先出”（FIFO，First In First Out）的原则。栈和队列广泛应用于计算机科学中的各种算法和问题求解中。

3.1 栈和队列的定义和特点

3.1.1 栈的定义和特点

栈（Stack）是一种只允许在一端进行插入和删除操作的数据结构，这一端叫做栈顶（Top）。栈的操作遵循“后进先出”（LIFO）原则，即最后插入的元素最先被删除。

栈的基本操作：

1. **Push**：将一个元素压入栈顶。
2. **Pop**：从栈顶移除一个元素。
3. **Peek** 或 **Top**：查看栈顶元素，但不移除它。
4. **isEmpty**：检查栈是否为空。

栈的特点：

- **后进先出**：最新插入的元素会最先被删除。
- 只能从栈顶进行操作，不能直接访问栈底的元素。
- 常用于解决递归问题、表达式求值、括号匹配等场景。

栈的应用：

- **函数调用管理**：程序的函数调用是通过栈来实现的，函数调用压栈，函数返回弹栈。
- **括号匹配问题**：可以利用栈来判断表达式中的括号是否匹配。

3.1.2 队列的定义和特点

队列（Queue）是一种允许从一端插入元素，从另一端删除元素的数据结构。队列的操作遵循“先进先出”（FIFO）原则，即最早插入的元素最先被删除。

队列的基本操作：

1. **Enqueue**：将一个元素加入到队列的尾部。
2. **Dequeue**：从队列的头部移除一个元素。
3. **Front**：查看队列头部的元素，但不移除它。
4. **isEmpty**：检查队列是否为空。

队列的特点：

- **先进先出**：最先插入的元素会最先被删除。
- 只能从队尾插入元素，从队头删除元素。
- 常用于任务调度、数据流控制、广度优先搜索等场景。

队列的应用：

- **任务调度**：操作系统中使用队列来管理任务的调度，确保任务按照顺序执行。
- **广度优先搜索（BFS）**：BFS 算法通常使用队列来实现，确保节点按层次顺序访问。

3.2 案例引入

为了更好地理解栈和队列的应用，下面通过两个经典的编程问题来引入和应用栈与队列。

1. **栈的应用案例：括号匹配**给定一个包含括号的表达式，判断括号是否匹配。我们可以使用栈来解决这个问题，遇到开括号时将其压栈，遇到闭括号时弹栈，如果弹栈时括号不匹配，则说明不合法。
2. **队列的应用案例：打印队列中的元素**假设我们有一个任务调度系统，任务按照队列的顺序执行。可以使用队列来模拟任务的添加与执行，保证任务按顺序完成。

接下来将通过 C++ 示例代码来实现这些应用案例。

C++ 代码实现：

1. 栈的应用：括号匹配

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

// 判断括号是否匹配
bool isValidParentheses(const string& s) {
    stack<char> st;

    for (char ch : s) {
        if (ch == '(' || ch == '[' || ch == '{') {
            st.push(ch);  // 遇到开括号，压栈
        } else if (ch == ')' || ch == ']' || ch == '}') {
            if (st.empty()) {
                return false;  // 遇到闭括号但栈为空，不匹配
            }
            char top = st.top();
            st.pop();
            // 检查是否匹配
            if ((ch == ')' && top != '(') ||
                (ch == ']' && top != '[') ||
                (ch == '}' && top != '{')) {
                return false;
            }
        }

    }
```

```
    return st.empty();  // 栈空则匹配成功，栈不空则有未匹配的括号
}

int main() {
    string expression = "{[()]}";
    if (isValidParentheses(expression)) {
        cout << "The parentheses are valid!" << endl;
    } else {
        cout << "The parentheses are invalid!" << endl;
    }
    return 0;
}
```

代码说明：

- 使用 `stack<char>` 来模拟栈操作。
- 遍历输入的字符串，当遇到开括号时将其压栈，遇到闭括号时弹栈并检查是否匹配。

输出示例：

```
The parentheses are valid!
```

2. 队列的应用：任务调度

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    // 创建队列
    queue<string> tasks;

    // 模拟任务调度
    tasks.push("Task 1");
    tasks.push("Task 2");
    tasks.push("Task 3");

    cout << "Executing tasks..." << endl;

    // 执行任务
    while (!tasks.empty()) {
        cout << "Executing: " << tasks.front() << endl;
        tasks.pop();  // 执行完任务后移除
    }

    return 0;
}
```

代码说明：

- 使用 `queue<string>` 来模拟任务队列。
- `push()` 用于将任务添加到队列的尾部，`pop()` 用于从队列头部移除任务。

输出示例：

```
Executing tasks...
Executing: Task 1
Executing: Task 2
Executing: Task 3
```

总结

- **栈**：后进先出（LIFO），主要应用于函数调用管理、表达式求值、括号匹配等场景。
- **队列**：先进先出（FIFO），主要应用于任务调度、广度优先搜索等场景。
- 栈和队列在数据结构和算法中具有广泛的应用，通过实际案例可以帮助更好地理解它们的特性和应用场景。

在实际问题中，栈和队列的选择取决于问题的性质，栈适合处理递归、回溯等问题，队列适合处理顺序或并行执行的任务。

3.3 栈的表示和操作的实现

栈作为一种线性数据结构，有两种常见的实现方式：顺序栈和链栈。顺序栈是通过数组来表示栈，而链栈则是通过链表来表示栈。在这部分，我们将详细介绍栈的类型定义、顺序栈和链栈的实现，并提供相关的 C++ 代码。

3.3.1 栈的类型定义

栈可以通过两种方式来实现：顺序栈和链栈。

- **顺序栈**：通过数组实现，栈的大小固定，元素存储在连续的内存空间中。
- **链栈**：通过链表实现，栈的大小可以动态变化，栈顶指针指向链表的头部。

对于栈的基本类型定义，我们首先需要定义栈的数据结构。栈的一般定义包括栈的大小、栈顶指针以及用于存储栈元素的数组或链表。

栈的类型定义：

```
// 栈的结构体定义
template <typename T>
struct Stack {
    T* data;  // 栈存储元素的数组或链表
    int top;   // 栈顶指针
    int capacity; // 栈的容量

    // 构造函数
    Stack(int size) {
        data = new T[size];  // 为数组分配内存
        top = -1;  // 初始化栈顶指针
        capacity = size;  // 设置栈的容量
    }


    // 析构函数
```

```
    ~Stack() {
        delete[] data;  // 释放栈的内存
    }

    // 栈的基本操作
    bool isEmpty() { return top == -1; }
    bool isFull() { return top == capacity - 1; }
    void push(const T& value);
    T pop();
    T peek();
};
```

在这个定义中，栈包含了 `data`（用于存储栈元素的数组）、`top`（栈顶指针）和 `capacity`（栈的容量）。接下来我们将分别介绍顺序栈和链栈的具体实现。

3.3.2 顺序栈的表示和实现

顺序栈是通过数组来实现的，因此需要事先确定栈的大小。在顺序栈中，栈顶元素的插入和删除操作都发生在数组的末尾。

顺序栈的实现：

```
#include <iostream>
using namespace std;

template <typename T>
class SeqStack {
private:
    T* data;   // 存储栈元素的数组
    int top;    // 栈顶指针
    int capacity;  // 栈的容量

public:
    // 构造函数
    SeqStack(int size) {
        data = new T[size];
        top = -1;  // 栈空时栈顶指针为 -1
        capacity = size;
    }

    // 析构函数
    ~SeqStack() {
        delete[] data;  // 释放栈的内存
    }

    // 判断栈是否为空
    bool isEmpty() {
        return top == -1;
    }


    // 判断栈是否已满
```

```
    bool isFull() {
        return top == capacity - 1;
    }

    // 入栈操作
    void push(const T& value) {
        if (isFull()) {
            cout << "Stack is full!" << endl;
            return;
        }
        data[++top] = value;  // 将元素压入栈顶
    }

    // 出栈操作
    T pop() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return T();  // 返回默认值
        }
        return data[top--];  // 弹出栈顶元素
    }

    // 获取栈顶元素
    T peek() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return T();  // 返回默认值
        }
        return data[top];  // 返回栈顶元素
    }

    // 打印栈中的所有元素
    void print() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return;
        }
        for (int i = 0; i <= top; i++) {
            cout << data[i] << " ";
        }
        cout << endl;
    }
};

int main() {
    SeqStack<int> stack(5);  // 创建一个容量为 5 的栈

    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.push(40);
    stack.push(50);
```

```
    stack.print();  // 打印栈中的所有元素

    cout << "Top element: " << stack.peek() << endl;  // 获取栈顶元素

    stack.pop();
    stack.print();  // 弹出栈顶元素并打印栈中元素

    return 0;
}
```

代码说明：

- `push()`：将元素压入栈顶，如果栈已满则输出提示信息。
- `pop()`：从栈顶弹出元素，如果栈为空则输出提示信息。
- `peek()`：返回栈顶元素，但不删除。
- `print()`：打印栈中的所有元素。

输出示例：

```
10 20 30 40 50
Top element: 50
10 20 30 40
```

3.3.3 链栈的表示和实现

链栈是一种使用链表实现的栈，它不需要预先定义容量，栈的大小可以动态扩展。每次入栈时，栈顶元素都会作为新节点加入链表。

链栈的实现：

```
#include <iostream>
using namespace std;

template <typename T>
struct Node {
    T data;         // 存储栈元素
    Node* next;     // 指向下一个节点
};

template <typename T>
class LinkStack {
private:
    Node<T>* top;   // 栈顶指针

public:
    // 构造函数
    LinkStack() : top(nullptr) {}

    // 析构函数
    ~LinkStack() {
        while (!isEmpty()) {
```

```
            pop();  // 弹出栈中的所有元素
        }
    }

    // 判断栈是否为空
    bool isEmpty() {
        return top == nullptr;
    }

    // 入栈操作
    void push(const T& value) {
        Node<T>* newNode = new Node<T>{value, top};  // 创建新节点
        top = newNode;  // 更新栈顶指针
    }

    // 出栈操作
    T pop() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return T();  // 返回默认值
        }
        Node<T>* temp = top;  // 保存栈顶节点
        T value = top->data;  // 获取栈顶元素
        top = top->next;  // 更新栈顶指针
        delete temp;  // 释放栈顶节点的内存
        return value;  // 返回栈顶元素
    }

    // 获取栈顶元素
    T peek() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return T();  // 返回默认值
        }
        return top->data;  // 返回栈顶元素
    }

    // 打印栈中的所有元素
    void print() {
        if (isEmpty()) {
            cout << "Stack is empty!" << endl;
            return;
        }
        Node<T>* current = top;
        while (current != nullptr) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};


int main() {
```

```
    LinkStack<int> stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.push(40);
    stack.push(50);

    stack.print();  // 打印栈中的所有元素

    cout << "Top element: " << stack.peek() << endl;  // 获取栈顶元素

    stack.pop();
    stack.print();  // 弹出栈顶元素并打印栈中元素

    return 0;
}
```

代码说明：

- 链栈使用一个链表节点 `Node` 来表示栈中的元素。每个节点包含数据和指向下一个节点的指针。
- `push()`：将新节点压入栈顶，更新栈顶指针。
- `pop()`：删除栈顶节点，并返回其数据。
- `peek()`：返回栈顶节点的数据，但不删除。
- `print()`：打印栈中的所有元素。

输出示例：

```
50 40 30 20 10
Top element: 50
40 30 20 10
```

总结

- **顺序栈**：通过数组

实现，栈的大小固定。适合用于容量已知且固定的情况。操作简单但可能浪费空间。

- **链栈**：通过链表实现，栈的大小动态变化。适合用于容量不确定的情况，不会浪费空间，但实现上相对复杂。

栈的应用广泛，包括函数调用管理、表达式求值、括号匹配、深度优先搜索等。顺序栈和链栈的选择依据具体应用场景而定。

3.4 栈与递归

递归是程序设计中的一种常见方法，它通过函数调用自身来解决问题。栈在递归过程中扮演了至关重要的角色，因为递归本质上就是一个栈操作过程。每一次递归调用都将当前函数的状态保存在栈中，并在递归结束时恢复这些状态。

在这一部分，我们将讨论递归与栈的关系，包括递归算法的应用、递归工作栈的作用、递归算法的效率分析以及如何通过栈将递归算法转换为非递归算法。

3.4.1 采用递归算法解决的问题

递归算法通过函数调用自身来简化问题的求解。许多经典问题都可以通过递归算法来高效解决。例如，计算阶乘、斐波那契数列、树的遍历、图的深度优先搜索等。

典型的递归问题：

1. **阶乘问题：**阶乘问题是递归算法的经典例子。`n! = n * (n-1)!`，递归的基本思想是将大问题拆解为小问题，直到问题达到最小的边界条件。

   阶乘递归算法的实现：

   ```
   #include <iostream>
   using namespace std;

   // 计算阶乘的递归函数
   int factorial(int n) {
       if (n == 0 || n == 1) {
           return 1;  // 递归的边界条件
       }
       return n * factorial(n - 1);  // 递归调用
   }

   int main() {
       int num = 5;
       cout << num << "! = " << factorial(num) << endl;
       return 0;
   }
   ```

   输出示例：

   ```
   5! = 120
   ```

2. **斐波那契数列：**斐波那契数列的定义是：`F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)`。这个问题也可以通过递归来解决。

   斐波那契数列的递归实现：

   ```
   #include <iostream>
   using namespace std;

   // 计算斐波那契数列的递归函数
   int fibonacci(int n) {
       if (n == 0) {
           return 0;  // 递归的边界条件
       }
       if (n == 1) {
           return 1;  // 递归的边界条件
       }
       return fibonacci(n - 1) + fibonacci(n - 2);  // 递归调用
   }

   int main() {
   ```

```
    int n = 6;
    cout << "Fibonacci of " << n << " is " << fibonacci(n) << endl;
    return 0;
}
```

输出示例：

```
Fibonacci of 6 is 8
```

3.4.2 递归过程与递归工作栈

递归函数的调用过程本质上是一个栈操作。每次递归调用时，程序会将当前函数的参数、局部变量以及执行状态（包括程序计数器等）压入栈中。当递归的某一层返回时，程序会从栈中弹出状态，并恢复到上一层。

递归过程中的栈：

1. 每当函数调用自身时，新的函数调用会被压入栈。
2. 当递归到达边界条件时，开始弹出栈，逐层返回结果。
3. 在递归过程中，每一层的状态都会被保存在栈中，直到递归调用结束。

以下是一个更具体的例子来说明递归过程中的栈操作：

递归过程中的栈演示：

```
#include <iostream>
using namespace std;

void recursiveFunction(int n) {
    cout << "Entering recursion with n = " << n << endl;

    // 递归终止条件
    if (n <= 0) {
        cout << "Base case reached" << endl;
        return;
    }

    // 递归调用
    recursiveFunction(n - 1);
    cout << "Returning from recursion with n = " << n << endl;
}

int main() {
    recursiveFunction(3);
    return 0;
}
```

输出示例：

```
Entering recursion with n = 3
Entering recursion with n = 2
Entering recursion with n = 1
Entering recursion with n = 0
Base case reached
Returning from recursion with n = 1
Returning from recursion with n = 2
Returning from recursion with n = 3
```

在上面的程序中，`recursiveFunction` 递归地调用自己，直到 `n == 0` 时停止。在每次递归调用时，当前函数的状态（即栈帧）会被压入栈，直到返回。

3.4.3 递归算法的效率分析

递归算法的效率往往依赖于递归调用的深度和每一层的工作量。递归过程的时间复杂度通常由递归公式决定。递归算法可能会导致大量的重复计算，尤其是存在重叠子问题时。

递归算法的效率分析：

1. **时间复杂度：**时间复杂度通常由递归调用的次数和每次调用的工作量决定。例如，在斐波那契数列的递归实现中，由于每个数值会被计算多次，因此效率非常低。递归树的大小和层数决定了算法的时间复杂度。
2. **空间复杂度：**递归的空间复杂度通常是栈的深度。例如，深度为 `n` 的递归会占用 `O(n)` 的空间。

示例：斐波那契数列的递归时间复杂度：

```
#include <iostream>
using namespace std;

// 计算斐波那契数列的递归函数
int fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);  // 递归调用
}

int main() {
    int n = 40;  // 输入较大的值，效率较低
    cout << "Fibonacci of " << n << " is " << fibonacci(n) << endl;
    return 0;
}
```

递归的时间复杂度为 O(2^n)，因为每个函数调用都会产生两个子函数调用。递归算法可以通过动态规划（记忆化递归）进行优化，减少重复计算。

3.4.4 利用栈将递归转换为非递归的方法

递归调用的栈实际上可以用显式的栈来模拟。在将递归算法转化为非递归算法时，可以使用栈来模拟递归调用的过程。例如，深度优先搜索（DFS）和树的遍历等问题可以通过显式栈来实现。

递归转非递归的示例：计算阶乘

递归的阶乘算法可以用栈来实现。我们通过显式栈来模拟函数调用的过程。

```
#include <iostream>
#include <stack>
using namespace std;

// 非递归计算阶乘的函数
int factorial(int n) {
    stack<int> st;  // 用栈来模拟递归过程
    int result = 1;

    // 将所有数字从 n 依次推入栈中
    while (n > 1) {
        st.push(n);
        n--;
    }

    // 从栈中逐个弹出，并进行乘法运算
    while (!st.empty()) {
        result *= st.top();  // 栈顶元素
        st.pop();  // 弹出栈顶元素
    }

    return result;
}

int main() {
    int num = 5;
    cout << num << "! = " << factorial(num) << endl;
    return 0;
}
```


输出示例：

```
5! = 120
```

在这个非递归实现中，我们用一个栈来存储每次的计算，并通过循环模拟递归的过程。这避免了递归调用时栈溢出的风险，同时提高了程序的可控性。

总结

- **递归算法**：通过函数自调用来分解问题，通常适合处理具有自相似结构的问题。
- **递归工作栈**：递归调用时，函数的状态会被压入栈，返回时恢复状态。栈的深度决定了递归的空间复杂度。
- **递归效率分析**：递归算法可能会存在重复计算，特别是在没有优化时。时间复杂度通常与递归的深度和分支数相关。
- **递归转非递归**：通过显式栈来模拟递归

调用的过程，从而避免栈溢出和提高控制力。

理解递归与栈的关系，以及如何将递归转换为非递归算法，对于解决复杂问题至关重要。

3.5 队列的表示和操作的实现

队列（Queue）是一种典型的线性数据结构，其遵循先进先出（FIFO, First In First Out）原则。队列有两种常见的表示方式：顺序表示和链式表示。顺序表示常通过数组实现，而链式表示则通过链表实现。

在本节中，我们将介绍队列的类型定义、顺序表示的循环队列实现，以及链式表示的链队列实现。

3.5.1 队列的类型定义

队列的基本操作包括：

1. **入队（Enqueue）**：将元素添加到队列的尾部。
2. **出队（Dequeue）**：从队列的头部移除元素。
3. **获取队头元素（Front/Peek）**：查看队列头部的元素，但不移除它。
4. **判断队列是否为空（isEmpty）**：检查队列是否有元素。
5. **判断队列是否已满（isFull）**：检查队列是否已达到最大容量。

队列可以通过两种方式表示：顺序表示和链式表示。下面是队列的基本类型定义。

```
template <typename T>
struct Queue {
    T* data;          // 存储队列元素的数组或链表
    int front;        // 队头指针
    int rear;         // 队尾指针
    int capacity;     // 队列的容量

    // 构造函数
    Queue(int size) {
        data = new T[size];
        front = rear = -1;  // 初始化队列为空
        capacity = size;
    }

    // 析构函数
    ~Queue() {
        delete[] data;
    }

    bool isEmpty() { return front == rear; }
    bool isFull() { return (rear + 1) % capacity == front; }
    void enqueue(const T& value);
    T dequeue();
    T frontElement();
};
```

3.5.2 循环队列 —— 队列的顺序表示和实现

循环队列是队列的一种特殊表示方法，它通过使用数组实现队列，且队列的尾部连接到头部。当队列满时，队尾指针会回绕到队头，实现数组空间的最大利用。

循环队列的特点：

- **队头指针**：指向队列中的第一个元素。
- **队尾指针**：指向队列中的最后一个元素的下一个位置。
- **数组空间利用**：使用模运算（`%`）来实现循环的效果。

循环队列的顺序实现：

```
#include <iostream>
using namespace std;

template <typename T>
class CircularQueue {
private:
    T* data;       // 存储队列元素的数组
    int front;     // 队头指针
    int rear;      // 队尾指针
    int capacity;  // 队列容量

public:
    // 构造函数
    CircularQueue(int size) {
        data = new T[size];
        front = rear = 0;  // 初始化队列为空
        capacity = size;
    }

    // 析构函数
    ~CircularQueue() {
        delete[] data;
    }

    // 判断队列是否为空
    bool isEmpty() {
        return front == rear;
    }

    // 判断队列是否已满
    bool isFull() {
        return (rear + 1) % capacity == front;
    }

    // 入队操作
    void enqueue(const T& value) {
        if (isFull()) {
            cout << "Queue is full!" << endl;
            return;
        }
        data[rear] = value;  // 将元素添加到队尾
        rear = (rear + 1) % capacity;  // 更新队尾指针
    }
```

```
    // 出队操作
    T dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return T();  // 返回默认值
        }
        T value = data[front];  // 获取队头元素
        front = (front + 1) % capacity;  // 更新队头指针
        return value;  // 返回出队的元素
    }

    // 获取队头元素
    T frontElement() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return T();  // 返回默认值
        }
        return data[front];  // 返回队头元素
    }

    // 打印队列中的元素
    void printQueue() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return;
        }
        int i = front;
        while (i != rear) {
            cout << data[i] << " ";
            i = (i + 1) % capacity;  // 采用模运算循环访问队列元素
        }
        cout << endl;
    }
};

int main() {
    CircularQueue<int> queue(5);  // 创建一个容量为 5 的循环队列

    queue.enqueue(10);
    queue.enqueue(20);
    queue.enqueue(30);
    queue.enqueue(40);
    queue.enqueue(50);

    queue.printQueue();  // 打印队列中的元素

    queue.dequeue();
    queue.printQueue();  // 弹出队头元素并打印队列中的元素

    queue.enqueue(60);  // 入队新元素
    queue.printQueue();  // 打印队列中的元素
```

```
    return 0;
}
```

输出示例：

```
10 20 30 40 50
20 30 40 50
20 30 40 50 60
```

代码说明：

- **入队（enqueue）**：将元素添加到队尾，并更新队尾指针。使用 `(rear + 1) % capacity` 来实现循环。
- **出队（dequeue）**：从队头移除元素，并更新队头指针。
- **队列为空检查（isEmpty）**：当队头指针与队尾指针相同，表示队列为空。
- **队列已满检查（isFull）**：当队尾指针的下一个位置与队头指针相同，表示队列已满。
- **打印队列**：从队头到队尾打印队列中的元素，使用 `(i + 1) % capacity` 来访问队列。

3.5.3 链队列 —— 队列的链式表示和实现

链队列是通过链表实现的队列，不需要事先确定大小。链队列的每个节点包含队列元素和指向下一个节点的指针。

链队列的特点：

- **队头指针**：指向队列中的第一个节点。
- **队尾指针**：指向队列中的最后一个节点。

链队列的实现：

```
#include <iostream>
using namespace std;

// 链表节点
template <typename T>
struct Node {
    T data;          // 队列元素
    Node* next;      // 指向下一个节点
};

// 链队列类
template <typename T>
class LinkQueue {
private:
    Node<T>* front;   // 队头指针
    Node<T>* rear;    // 队尾指针

public:
    // 构造函数
    LinkQueue() {
        front = rear = nullptr;  // 初始化队列为空
    }

    // 析构函数
```

```
    ~LinkQueue() {
        while (!isEmpty()) {
            dequeue();  // 释放队列中的所有节点
        }
    }

    // 判断队列是否为空
    bool isEmpty() {
        return front == nullptr;
    }

    // 入队操作
    void enqueue(const T& value) {
        Node<T>* newNode = new Node<T>{value, nullptr};  // 创建新节点
        if (isEmpty()) {
            front = rear = newNode;  // 队列为空时，新节点既是队头也是队尾
        } else {
            rear->next = newNode;  // 队尾节点的 next 指向新节点
            rear = newNode;        // 更新队尾指针
        }
    }

    // 出队操作
    T dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return T();  // 返回默认值
        }
        Node<T>* temp = front;  // 保存队头节点
        T value = front->data;  // 获取队头元素
        front = front->next;    // 更新队头指针
        if (front == nullptr) { // 队列为空时，更新队尾指针
            rear = nullptr;
        }
        delete temp;  // 释放队头节点
        return value;  // 返回出队的元素
    }

    // 获取队头元素
    T frontElement() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
            return T();  // 返回默认值
        }
        return front->data;
```



```
// 返回队头元素}
// 打印队列中的元素
void printQueue() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
```

```
        return;
    }
    Node<T>* temp = front;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
};

int main() {LinkQueue queue;  // 创建链队列
```

```
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);
queue.enqueue(40);

queue.printQueue();  // 打印队列中的元素

queue.dequeue();
queue.printQueue();  // 弹出队头元素并打印队列中的元素

return 0;
}
```


输出示例：

```
10 20 30 4020 30 40
```

代码说明：

- **入队（enqueue）**：创建新节点，并将其添加到队列的尾部。如果队列为空，则新节点同时是队头和队尾。
- **出队（dequeue）**：移除队头节点，并返回该节点的值。如果队列为空，则输出提示。
- **打印队列**：从队头开始，逐一访问链表中的元素。

总结

1. **循环队列**：通过数组实现，队列的尾部连接到头部，有效地利用了数组空间。适合用于固定容量的队列应用。
2. **链队列**：通过链表实现，队列大小动态变化，适合用于容量不确定的队列应用。

第 4 章 串、数组和广义表

在本章中，我们将探讨与字符串、数组以及广义表相关的内容。我们将从基本概念出发，逐步深入，介绍它们的定义、存储结构以及常见操作，尤其是如何处理字符串相关的算法和技术。

4.1 串的定义

串（String）是由零个或多个字符组成的有限序列，通常用于表示文本数据。在计算机中，串是字符的有序集合，通常用一维数组或链表来表示。

串的基本操作包括：

1. **串的连接（Concatenation）**：将两个串合并成一个串。
2. **串的长度（Length）**：获取串中字符的个数。
3. **串的比较（Comparison）**：判断两个串是否相等，或者哪个串更大。
4. **串的模式匹配（Pattern Matching）**：查找子串的位置。

在一些编程语言中，串是一个数据类型，如Python中的`str`类型，而在C/C++中，串通常由字符数组表示。

4.2 案例引入

假设我们有一个文本文件，其中包含一些句子和单词。我们想要实现一个简单的应用程序，该程序能够实现以下功能：

1. **统计单词出现的次数。**
2. **在字符串中查找特定的单词。**
3. **替换字符串中的某些部分。**

为了完成这些任务，我们需要对字符串的存储结构和操作有一个深入的理解。

4.3 串的类型定义、存储结构及其运算

在本节中，我们将详细介绍字符串的抽象类型定义、存储结构以及常见的字符串运算。

4.3.1 串的抽象类型定义

串作为一种数据类型，其抽象定义包括以下几个方面：

- **定义**：串是由若干字符组成的有限序列。
- 基本操作
  ：
  - **长度（Length）**：计算串的长度。
  - **连接（Concatenate）**：将两个串连接成一个新的串。
  - **获取字符**：获取串中的某个字符。
  - **模式匹配**：查找子串的位置。

C++中可以通过类来实现一个串的抽象数据类型。以下是一个简单的串的类定义：

```
#include <iostream>
#include <cstring>
using namespace std;
```

```
class String {
private:
    char* str;  // 存储字符串的指针
    int length; // 字符串长度

public:
    // 构造函数
    String(const char* s) {
        length = strlen(s);
        str = new char[length + 1];  // 为字符串分配内存
        strcpy(str, s);  // 复制字符串
    }

    // 析构函数
    ~String() {
        delete[] str;  // 释放分配的内存
    }

    // 获取字符串长度
    int getLength() const {
        return length;
    }

    // 获取字符串内容
    const char* getString() const {
        return str;
    }

    // 串的连接操作
    String concatenate(const String& other) {
        int newLength = length + other.getLength();
        char* newStr = new char[newLength + 1];
        strcpy(newStr, str);
        strcat(newStr, other.getString());  // 连接两个字符串
        return String(newStr);
    }

    // 串的比较操作
    bool compare(const String& other) {
        return strcmp(str, other.getString()) == 0;  // 字符串比较
    }

    // 打印字符串
    void print() const {
        cout << str << endl;
    }
};

int main() {
    String s1("Hello");
    String s2(" World");
    String s3 = s1.concatenate(s2);
```

```
    s1.print();  // 输出: Hello
    s2.print();  // 输出:  World
    s3.print();  // 输出: Hello World

    return 0;
}
```

代码说明：

- `String` 类定义了字符串的基本操作，如获取长度、连接字符串、比较两个字符串等。
- `concatenate()` 方法将两个字符串连接成一个新的字符串。
- `compare()` 方法比较两个字符串是否相等。

4.3.2 串的存储结构

在计算机中，串通常通过以下两种方式存储：

1. **顺序存储（数组）**：
   - 串可以通过一个字符数组存储，字符数组的长度表示串的长度，数组的每个元素表示串中的一个字符。
   - 优点：存储和访问方便。
   - 缺点：无法动态增长，空间利用率不高，尤其是对于动态长度的字符串。
2. **链式存储（链表）**：
   - 通过链表来存储字符串，其中每个节点存储一个字符和指向下一个节点的指针。
   - 优点：能够动态地存储字符串，适合处理变长字符串。
   - 缺点：访问较慢，空间开销大。

顺序存储的示例：

```
char str[] = "Hello, World!";
```

链式存储的示例：

```
struct Node {
    char data;
    Node* next;
};

Node* createNode(char c) {
    Node* newNode = new Node();
    newNode->data = c;
    newNode->next = nullptr;
    return newNode;
}
```

4.3.3 串的模式匹配算法

模式匹配算法的目的是在一个串（文本串）中查找另一个串（模式串）的出现位置。常见的模式匹配算法包括：

1. 朴素算法

   ：

   - 通过从文本串的每个位置开始，逐个字符地与模式串进行比较。
   - 时间复杂度：O(n * m)，其中n是文本串的长度，m是模式串的长度。

```
int naiveMatch(const char* text, const char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);

    for (int i = 0; i <= n - m; ++i) {
        int j = 0;
        while (j < m && text[i + j] == pattern[j]) {
            ++j;
        }
        if (j == m) return i;  // 找到匹配，返回位置
    }
    return -1;  // 未找到匹配
}
```

1. KMP算法

   (Knuth-Morris-Pratt算法)：

   - 通过预处理模式串，构造一个部分匹配表，避免重复比较。
   - 时间复杂度：O(n + m)，大大优化了朴素算法。

```
void computePrefixFunction(const char* pattern, int* pi) {
    int m = strlen(pattern);
    pi[0] = 0;
    for (int i = 1, j = 0; i < m; ++i) {
        while (j > 0 && pattern[i] != pattern[j]) {
            j = pi[j - 1];
        }
        if (pattern[i] == pattern[j]) {
            ++j;
        }
        pi[i] = j;
    }
}

int KMPMatch(const char* text, const char* pattern) {
    int n = strlen(text);
    int m = strlen(pattern);
    int* pi = new int[m];
    computePrefixFunction(pattern, pi);

    for (int i = 0, j = 0; i < n; ++i) {
        while (j > 0 && text[i] != pattern[j]) {
            j = pi[j - 1];
        }
        if (text[i] == pattern[j]) {
            ++j;
```

```
        }
        if (j == m) {
            delete[] pi;
            return i - m + 1;  // 返回匹配位置
        }
    }
    delete[] pi;
    return -1;  // 未找到匹配
}
```

小结

在本章中，我们深入探讨了串的定义、存储结构和常见操作。通过学习串的抽象类型定义、存储方式以及模式匹配算法，我们掌握了如何有效地处理和操作字符串数据。在实际应用中，选择合适的存储结构和算法可以大大提高程序的性能和可扩展性。

4.4 数组

数组（Array）是一种线性数据结构，用于存储一组相同类型的元素。数组在内存中是连续存储的，可以通过索引直接访问任何一个元素。在本节中，我们将探讨数组的类型定义、顺序存储方式以及特殊矩阵的压缩存储。

4.4.1 数组的类型定义

数组的基本特点是能够存储多个同类型元素，并通过索引（下标）访问这些元素。数组在大多数编程语言中都有一种类似的定义方式。

数组的定义：

- **一维数组**：一维数组是最简单的数组类型，可以视为一个元素线性排列的集合。
- **多维数组**：二维数组或三维数组是更复杂的数组，通常用于表示表格数据或更高维的数据结构。

C++ 中的数组定义：

1. **一维数组**：

```
int arr[5];  // 定义一个包含5个整数元素的一维数组
```

1. **二维数组**：

```
int arr[3][4];  // 定义一个3行4列的二维数组
```

1. **动态数组**：

```
int* arr = new int[5];  // 动态分配一个包含5个整数元素的一维数组
```

1. **数组初始化**：

```
int arr[5] = {1, 2, 3, 4, 5};  // 定义并初始化一维数组
int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};  // 初始化二维数组
```

4.4.2 数组的顺序存储

数组的**顺序存储**是指数组中的元素在内存中是连续存储的，这使得数组在访问时具有较高的效率。顺序存储结构是数组的最基本存储方式，适用于元素个数固定且需要高效访问的场景。

数组顺序存储的特点：

1. **访问速度快**：由于数组元素在内存中是连续存储的，可以通过下标直接访问任意元素，时间复杂度为 O(1)。
2. **固定大小**：数组的大小在声明时确定，无法动态调整（对于静态数组而言）。
3. **内存连续性**：数组的所有元素在内存中是紧密排列的，减少了内存碎片，但也可能造成内存的浪费。
4. **插入删除效率低**：数组中的元素插入和删除操作通常需要移动大量数据，时间复杂度为 O(n)。

顺序存储的操作：

1. **访问数组元素**：通过数组下标直接访问元素。

   ```
   int arr[5] = {1, 2, 3, 4, 5};
   int x = arr[2];  // 访问数组中下标为 2 的元素，值为 3
   ```

2. **遍历数组**：通过循环遍历数组中的元素。

   ```
   for (int i = 0; i < 5; ++i) {
       cout << arr[i] << " ";
   }
   ```

3. **修改数组元素**：

   ```
   arr[3] = 10;  // 修改数组中下标为 3 的元素，值为 10
   ```

4. **多维数组的顺序存储**：
   - 多维数组在内存中是按照**行优先**的顺序存储的，即先存储第一行，后存储第二行，依此类推。
   - 例如，二维数组 `arr[3][4]` 将会按顺序存储为一维数组 `arr[0][0], arr[0][1], arr[0][2], arr[0][3], arr[1][0], ...`。

   ```
   int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
   // 内存中存储顺序为：1 2 3 4 5 6
   ```

数组的存储示意图：

```
int arr[4] = {10, 20, 30, 40};
// 内存布局（假设数组在内存的起始地址为1000）：
地址 1000 | 1004 | 1008 | 1012
值     10   |  20  |  30  |  40
```

4.4.3 特殊矩阵的压缩存储

特殊矩阵是指某些矩阵具有稀疏性，即矩阵中的大部分元素为零。为了节省存储空间，可以使用**压缩存储**技术，仅保存非零元素及其位置。这种技术通常用于**稀疏矩阵**的存储，能够大幅减少内存开销。

常见的压缩存储方法：

1. **压缩行存储（CSR, Compressed Sparse Row）**：
   - 只存储非零元素的值、每行非零元素的开始位置以及每个非零元素所在的列索引。
   - 适用于按行访问较频繁的稀疏矩阵。
2. **压缩列存储（CSC, Compressed Sparse Column）**：
   - 类似于CSR，但存储的是列索引而不是行索引。
   - 适用于按列访问较频繁的稀疏矩阵。
3. **坐标列表（COO, Coordinate List）**：
   - 存储矩阵的非零元素以及它们的行和列坐标，适用于某些稀疏矩阵的快速操作。

例：压缩行存储（CSR）

假设我们有如下稀疏矩阵：

```
0  0  0  3  0
0  0  0  0  0
0  0  0  0  2
5  0  0  0  0
```

对于这个矩阵，我们可以用压缩行存储来保存其非零元素。

1. **非零元素的值**：`3, 2, 5`
2. **每行非零元素的开始位置**：`0, 3, 4, 5`（分别表示每行的第一个非零元素在值数组中的位置）
3. **每个非零元素所在的列索引**：`3, 4, 0`（表示每个非零元素所在的列）

```
// CSR存储格式：
int values[] = {3, 2, 5};       // 非零元素值
int columnIndices[] = {3, 4, 0};  // 非零元素列索引
int rowPointers[] = {0, 3, 4, 5};  // 每行非零元素的开始位置

// 稀疏矩阵的压缩存储可以节省大量内存空间
```

代码示例：使用CSR压缩存储稀疏矩阵

```
#include <iostream>
using namespace std;

class SparseMatrix {
private:
    int* values;         // 非零元素值
    int* columnIndices;  // 列索引
    int* rowPointers;    // 行指针

    int numRows, numCols, numNonZero;
```

```
public:
    SparseMatrix(int rows, int cols, int nonZero) {
        numRows = rows;
        numCols = cols;
        numNonZero = nonZero;

        values = new int[nonZero];
        columnIndices = new int[nonZero];
        rowPointers = new int[rows + 1];
    }

    ~SparseMatrix() {
        delete[] values;
        delete[] columnIndices;
        delete[] rowPointers;
    }

    // 设置矩阵的值、列索引和行指针
    void setData(int* vals, int* cols, int* rows) {
        for (int i = 0; i < numNonZero; ++i) {
            values[i] = vals[i];
            columnIndices[i] = cols[i];
        }
        for (int i = 0; i <= numRows; ++i) {
            rowPointers[i] = rows[i];
        }
    }

    // 打印压缩存储的矩阵
    void print() {
        cout << "Values: ";
        for (int i = 0; i < numNonZero; ++i) {
            cout << values[i] << " ";
        }
        cout << endl;

        cout << "Column Indices: ";
        for (int i = 0; i < numNonZero; ++i) {
            cout << columnIndices[i] << " ";
        }
        cout << endl;

        cout << "Row Pointers: ";
        for (int i = 0; i <= numRows; ++i) {
            cout << rowPointers[i] << " ";
        }
        cout << endl;
    }
  };

int main() {int values[] = {3, 2, 5};int columnIndices[] = {3, 4, 0};int rowPointers[] = {0, 3,
4, 5};
```

```
SparseMatrix mat(4, 5, 3);
mat.setData(values, columnIndices, rowPointers);
mat.print();

return 0;

}
```

输出示例：

```
Values: 3 2 5Column Indices: 3 4 0Row Pointers: 0 3 4 5
```

小结

在本节中，我们深入探讨了数组的类型定义、顺序存储方式以及稀疏矩阵的压缩存储方法。通过顺序存储和压缩存储技术，我们可以有效地存储和访问不同类型的数据，特别是在处理稀疏矩阵时，压缩存储方法能够显著节省内存。

4.5 广义表

广义表（Generalized List，简称GL）是一种递归的数据结构，可以看作是一个包含原子元素和/或子表的表。在广义表中，表的元素不仅可以是数据项（如整数、字符等），还可以是其他广义表，形成一种递归嵌套结构。广义表是一种扩展了线性表（如数组、链表）概念的数据结构。

广义表常用于 Lisp 等编程语言中，支持数据的动态结构变化。其灵活性使得它能够表示复杂的树形结构，适用于表达各种递归结构的数据，如解析表达式、树形结构的表示等。

在本节中，我们将介绍广义表的定义、存储结构，并给出相关的实现示例。

4.5.1 广义表的定义

广义表可以定义为：

- **原子元素**：例如，整数、字符等基本数据。
- **子表**：包含其他广义表作为元素的表。

广义表的定义具有递归性质：一个广义表可以包含多个元素，而这些元素可以是原子元素，也可以是其他广义表。广义表可以是**非空表**（包含一个或多个元素）或**空表**（通常用一个空指针表示）。

广义表的表示方式通常是递归的，因为一个表的元素可能是另一个表。

广义表的例子：

- 空表：`()` 或者 `nil`

- 非空广义表：`(1, 2, (3, 4), 5)`，这个广义表包含了原子元素 1, 2, 5 和一个子表 `(3, 4)`。

广义表的操作：

- **首元素（first）**：获取广义表的第一个元素。
- **剩余部分（rest）**：获取广义表中去掉第一个元素后的部分。
- **判断是否为空表（null）**：判断一个广义表是否为空。

广义表的递归性质使得它非常适合用链表来实现。

4.5.2 广义表的存储结构

广义表的存储结构是递归的，每个广义表的元素要么是原子元素（如整数、字符），要么是另一个广义表。为此，我们可以通过链表来实现广义表。链表节点可以包含两部分：一个存储元素的指针，另一个是指向下一个元素的指针。如果当前元素是一个子表，它可以指向一个新的广义表。

广义表的链式存储结构：

1. **原子元素**：通常表示为基本数据类型。
2. **子表**：也是一个广义表，因此是递归结构。

广义表的链式存储可以采用以下方式实现：

- 每个节点可以包含一个**数据域**（存储元素）和一个**指针域**（指向下一个节点或子表）。
- 对于子表，指针指向另一个广义表。

广义表节点定义：


```
#include <iostream>
using namespace std;

// 定义广义表节点
class GList {
public:
    bool isAtom;  // 判断元素是否为原子元素
    union {
        int atom;  // 如果是原子元素，存储数据
        GList* sublist;  // 如果是子表，指向另一个广义表
    };
    GList* next;  // 指向下一个元素

    // 构造原子元素
    GList(int value) {
        isAtom = true;
        atom = value;
        next = nullptr;
    }

    // 构造子表
    GList(GList* list) {
        isAtom = false;
        sublist = list;
        next = nullptr;
```

```
    }

    // 添加下一个元素
    void addNext(GList* nextElement) {
        next = nextElement;
    }
};

// 打印广义表
void printGList(GList* list) {
    if (list == nullptr) {
        cout << "()";
        return;
    }

    cout << "(";
    while (list != nullptr) {
        if (list->isAtom) {
            cout << list->atom;
        } else {
            printGList(list->sublist);
        }
        if (list->next != nullptr) {
            cout << ", ";
        }
        list = list->next;
    }
    cout << ")";
}

int main() {
    // 构建广义表 (1, 2, (3, 4), 5)
    GList* g1 = new GList(1);
    GList* g2 = new GList(2);
    GList* g3 = new GList(3);
    GList* g4 = new GList(4);
    GList* g5 = new GList(5);

    // 创建子表 (3, 4)
    g3->addNext(g4);
    GList* sublist = new GList(g3);

    // 构建整个广义表
    g1->addNext(g2);
    g2->addNext(sublist);
    sublist->addNext(g5);

    // 打印广义表
    printGList(g1);  // 输出: (1, 2, (3, 4), 5)

    return 0;
}
```

代码解释：

- **GList类**：表示广义表的节点。它包含两个主要部分：
  - `isAtom`：用于判断当前节点是否为原子元素。
  - `union`：使用 `union` 来存储原子元素或子表。
  - `next`：指向下一个节点。
- **构造函数**：
  - 如果是原子元素，则将 `atom` 设置为元素值。
  - 如果是子表，则将 `sublist` 设置为指向子表的指针。
- **addNext**：用于将下一个元素连接到当前节点。
- **printGList**：递归打印广义表。它会遍历广义表的每一个元素，如果元素是原子，则直接打印；如果是子表，则递归调用 `printGList` 打印子表。

递归结构的好处

广义表的一个重要特点就是其递归结构，这使得它非常适合用链表来表示。每个元素的 `next` 指针可以指向下一个元素或者子表。通过递归处理子表，广义表能够灵活地表示和处理树形结构、层次结构等复杂数据。

广义表的操作可以通过递归函数来实现，例如：

1. 获取广义表的首元素（first）。
2. 获取广义表的剩余部分（rest）。
3. 判断广义表是否为空。

这种递归处理的方式非常简洁，并且易于扩展。

小结

在本节中，我们讨论了广义表的定义、存储结构以及如何在 C++ 中实现广义表。广义表是一个递归的数据结构，可以包含原子元素和子表，通过链表存储来实现。通过递归操作，广义表能够灵活地表示各种层次结构和树形结构，是处理复杂数据结构的强大工具。

第 5 章 树和二叉树

树和二叉树是两种非常重要的非线性数据结构，它们广泛应用于各种领域，如计算机图形学、数据库管理、文件系统等。在本章中，我们将详细介绍树和二叉树的定义、基本术语以及如何在 C++ 中实现这些数据结构。

5.1 树和二叉树的定义

树是由节点组成的非线性数据结构，每个节点包含数据元素和指向子节点的指针。树的应用广泛，如文件系统、决策树等。二叉树是树的一种特殊形式，其中每个节点最多有两个子节点。

5.1.1 树的定义

树（Tree） 是一种由若干节点组成的集合，并满足以下两个条件：

1. **根节点**：树有一个特定的节点称为根节点。
2. **节点**：每个节点包含一个数据元素，并且可以有多个子节点。

树是一个递归结构，根节点下面可以有若干个子树，这些子树本身也是树。树的基本特点是每个节点都可能有多个子节点，但通常没有循环。

树的基本术语：

1. **根节点（Root）**：树的最上层节点。
2. **节点（Node）**：树中的元素，每个节点包含数据和指向子节点的指针。
3. **父节点（Parent）**：节点直接连接的上级节点。
4. **子节点（Child）**：一个节点的直接后代。
5. **叶节点（Leaf）**：没有子节点的节点。
6. **高度（Height）**：树的最大层数。
7. **深度（Depth）**：节点到根节点的路径长度。
8. **兄弟节点（Sibling）**：同一个父节点的其他节点。

树的广泛应用包括：

- **文件系统**：文件夹包含子文件夹和文件，每个文件夹都是一个子树。
- **XML 文档**：文档的元素可以形成树状结构。
- **决策树**：机器学习中的决策树用于分类任务。

5.1.2 树的基本术语

除了根节点、父节点、子节点等基本术语外，树的深度、高度以及节点的度（每个节点的子节点个数）也是树的基本术语。

- **度（Degree）**：一个节点的子节点数量。
- **树的深度（Depth of a tree）**：从根节点到叶节点的最长路径上的边数。
- **树的高度（Height of a tree）**：树中所有节点的最大深度。

例：树的示意图

```
        A           <- 根节点
      /   \
     B     C        <- 子节点
    / \   / \
   D   E F   G      <- 叶节点
```

- 根节点：A
- A的子节点：B, C
- B的子节点：D, E
- C的子节点：F, G
- 叶节点：D, E, F, G
- 高度：3（根节点A到叶节点的最长路径）
- 深度：根节点深度为0，节点B的深度为1，节点D的深度为2

5.1.3 二叉树的定义

二叉树（Binary Tree） 是一种特殊的树，其中每个节点最多只能有两个子节点，通常这两个子节点被称为**左子节点**和**右子节点**。二叉树是一种非常重要的数据结构，广泛应用于表达式解析、搜索树等领域。

二叉树的定义：

- 每个节点最多有两个子节点。
- 每个节点包含一个数据元素及指向其左、右子节点的指针。

二叉树的应用包括：

- **二叉查找树（BST）**：一种特殊的二叉树，用于快速查找、插入和删除操作。
- **堆（Heap）**：二叉树的一种特殊形式，用于实现优先队列。

二叉树的性质：

1. **满二叉树（Full Binary Tree）**：每个节点要么是叶子节点，要么有两个子节点。
2. **完全二叉树（Complete Binary Tree）**：除了最后一层，其他每一层都是满的，且最后一层的节点都在左侧。
3. **平衡二叉树（Balanced Binary Tree）**：任何节点的左右子树的高度差不超过1。

5.2 案例引入

在了解了树和二叉树的定义及其基本术语后，接下来我们将通过 C++ 实现一个简单的树和二叉树。通过代码示例，您将能够更好地理解这些数据结构的实际应用。

代码示例：树的实现

我们将首先实现一个简单的树结构，然后实现二叉树。

```
#include <iostream>
#include <vector>
using namespace std;

// 树的节点定义
class TreeNode {
public:
    int data;              // 节点的数据
    vector<TreeNode*> children; // 子节点列表

    TreeNode(int value) : data(value) {} // 构造函数

    // 添加子节点
    void addChild(TreeNode* child) {
        children.push_back(child);
    }
};

// 打印树的结构
void printTree(TreeNode* root, int level = 0) {
    if (root == nullptr) return;
```

```
    // 打印当前节点的值
    for (int i = 0; i < level; ++i) cout << "  "; // 缩进
    cout << root->data << endl;

    // 打印所有子节点
    for (TreeNode* child : root->children) {
        printTree(child, level + 1);
    }
}

int main() {
    // 创建节点
    TreeNode* root = new TreeNode(1);  // 根节点
    TreeNode* child1 = new TreeNode(2);
    TreeNode* child2 = new TreeNode(3);
    TreeNode* child3 = new TreeNode(4);

    // 构建树
    root->addChild(child1);
    root->addChild(child2);
    child1->addChild(child3);

    // 打印树结构
    printTree(root);

    return 0;
}
```

输出：

```
1
  2
    4
  3
```

代码解释：

- `TreeNode` 类：表示树的节点，包含数据和一个子节点列表。
- `addChild` 方法：将子节点添加到当前节点的子节点列表中。
- `printTree` 函数：递归打印树的结构，使用缩进表示树的层次。

代码示例：二叉树的实现

接下来，我们将实现一个简单的二叉树，支持插入节点和遍历操作。

```
#include <iostream>
using namespace std;

// 二叉树节点定义
class BinaryTreeNode {
public:
```

```
    int data;                    // 节点的数据
    BinaryTreeNode* left;        // 左子节点
    BinaryTreeNode* right;       // 右子节点

    BinaryTreeNode(int value) : data(value), left(nullptr), right(nullptr) {} // 构造函数
};

// 前序遍历
void preOrderTraversal(BinaryTreeNode* root) {
    if (root == nullptr) return;
    cout << root->data << " ";  // 访问根节点
    preOrderTraversal(root->left);  // 递归遍历左子树
    preOrderTraversal(root->right); // 递归遍历右子树
}

// 中序遍历
void inOrderTraversal(BinaryTreeNode* root) {
    if (root == nullptr) return;
    inOrderTraversal(root->left);   // 递归遍历左子树
    cout << root->data << " ";      // 访问根节点
    inOrderTraversal(root->right);  // 递归遍历右子树
}

// 后序遍历
void postOrderTraversal(BinaryTreeNode* root) {
    if (root == nullptr) return;
    postOrderTraversal(root->left);  // 递归遍历左子树
    postOrderTraversal(root->right); // 递归遍历右子树
    cout << root->data << " ";       // 访问根节点
}

int main() {
    // 创建节点
    BinaryTreeNode* root = new BinaryTreeNode(1);
    root->left = new BinaryTreeNode(2);
    root->right = new BinaryTreeNode(3);
    root->left->left = new BinaryTreeNode(4);
    root->left->right = new BinaryTreeNode(5);

    // 打印二叉树的遍历
    cout << "Preorder Traversal: ";
    preOrderTraversal(root);
    cout << endl;

    cout << "Inorder Traversal: ";
    inOrderTraversal(root);
    cout << endl;

    cout << "Postorder Traversal: ";
    postOrderTraversal(root);
    cout << endl;


    return 0;
```

```
    }
```

输出：

```
Preorder Traversal: 1 2 4 5 3Inorder Traversal: 4 2 5 1 3Postorder Traversal: 4 5 2 3 1
```

代码解释：

- `BinaryTreeNode` 类：表示二叉树的节点，包含数据、左子节点和右子节点。
- `preOrderTraversal`：前序遍历，访问根节点后递归遍历左右子树。
- `inOrderTraversal`：中序遍历，递归遍历左子树后访问根节点，再递归遍历右子树。
- `postOrderTraversal`：后序遍历，递归遍历左右子树后访问根节点。

小结

本章介绍了树和二叉树的定义及其基本术语，并通过 C++ 实现了树和二叉树的数据结构。树是一种非线性的数据结构，广泛应用于文件系统、决策树等领域；二叉树是一种特殊的树，其中每个节点最多有两个子节点。通过实现二叉树的插入和遍历操作，我们能够更好地理解这些数据结构的工作原理。

5.3 树和二叉树的抽象数据类型定义

抽象数据类型（Abstract Data Type，简称ADT）是指数据的定义及其操作的集合。ADT描述了数据的逻辑结构和可以对数据进行的操作，但不涉及数据的具体实现。在本节中，我们将定义树和二叉树的抽象数据类型，并详细讨论它们的操作。

5.3.1 树的抽象数据类型（ADT）

树的抽象数据类型包括节点、子树、根节点等元素及其相关操作。以下是树的基本操作：

1. **创建树**：
   - `Tree create()`：创建一棵新的空树。
2. **插入节点**：
   - `void insert(TreeNode* parent, TreeNode* child)`：向指定的父节点插入一个新的子节点。
3. **删除节点**：
   - `void delete(TreeNode* node)`：删除指定的节点。
4. **获取树的根节点**：
   - `TreeNode* root()`：返回树的根节点。
5. **遍历树**：
   - `void traverse(TreeNode* node)`：遍历树的所有节点，通常为前序、后序或中序遍历。
6. **判断树是否为空**：
   - `bool isEmpty()`：判断树是否为空。
7. **获取节点的子节点**：

- `List<TreeNode*> children(TreeNode* node)`：返回一个节点的所有子节点。

8. **获取节点的父节点**：
   - `TreeNode* parent(TreeNode* node)`：返回指定节点的父节点。

5.3.2 二叉树的抽象数据类型（ADT）

二叉树是一种特殊类型的树，其中每个节点最多有两个子节点（通常称为左子节点和右子节点）。二叉树的抽象数据类型包含以下操作：

1. **创建二叉树**：
   - `BinaryTree create()`：创建一棵空的二叉树。
2. **插入节点**：
   - `void insert(BinaryTreeNode* parent, BinaryTreeNode* left, BinaryTreeNode* right)`：向指定的父节点插入左子节点和右子节点。
3. **删除节点**：
   - `void delete(BinaryTreeNode* node)`：删除指定节点。
4. **获取根节点**：
   - `BinaryTreeNode* root()`：返回二叉树的根节点。
5. **遍历二叉树**：
   - `void preOrderTraversal(BinaryTreeNode* node)`：前序遍历。
   - `void inOrderTraversal(BinaryTreeNode* node)`：中序遍历。
   - `void postOrderTraversal(BinaryTreeNode* node)`：后序遍历。
6. **判断二叉树是否为空**：
   - `bool isEmpty()`：判断二叉树是否为空。
7. **获取节点的左子节点**：
   - `BinaryTreeNode* leftChild(BinaryTreeNode* node)`：获取节点的左子节点。
8. **获取节点的右子节点**：
   - `BinaryTreeNode* rightChild(BinaryTreeNode* node)`：获取节点的右子节点。

5.4 二叉树的性质和存储结构

在这一节中，我们将讨论二叉树的性质及其常见的存储结构。二叉树是数据结构中的重要类型，具有多种应用，如在搜索树、堆等领域中都有广泛应用。

5.4.1 二叉树的性质

二叉树具有一些基本的性质，了解这些性质有助于更好地理解和实现二叉树。以下是二叉树的主要性质：

1. **节点数与高度**：
   - 一个高度为 `h` 的二叉树，最多有 `2^h - 1` 个节点。
   - 二叉树的高度是树中从根节点到最深节点的最长路径上的边数。
2. **叶子节点数与总节点数**：

- 假设二叉树有 `n` 个节点，其中 `L` 个是叶子节点，那么二叉树的内部节点数（非叶节点）为 `n - L`。而根据性质，叶子节点 `L` 与非叶节点之间的关系为：
  `L = (n + 1) / 2`。这说明，在一棵完全二叉树中，叶子节点的个数约为总节点数的一半。

3. **完全二叉树的性质**：
   - 完全二叉树是一种特殊的二叉树，其中除了最底层之外，其他每一层的节点数都达到最大值，并且最底层的节点都集中在左侧。
   - 完全二叉树的节点数为 `n` 时，其高度为 $\lfloor \log_2 n \rfloor$。
4. **满二叉树的性质**：
   - 满二叉树是一种特殊的二叉树，其中每个节点都具有两个子节点，且所有叶子节点都在同一层。
   - 满二叉树的节点数与高度之间有着明确的关系。假设满二叉树的高度为 `h`，则节点数为 `2^h - 1`。
5. **二叉树的平衡性**：
   - 平衡二叉树（如 AVL 树、红黑树）是一类特殊的二叉树，它要求任何节点的左右子树高度差不超过 1。平衡二叉树的高度通常是 `O(log n)`，这使得它们在查找、插入、删除等操作时能够保持较好的效率。
6. **二叉树的遍历**：
   - 二叉树有三种基本遍历方式：前序遍历、中序遍历和后序遍历。
     - **前序遍历**：根节点 → 左子树 → 右子树
     - **中序遍历**：左子树 → 根节点 → 右子树
     - **后序遍历**：左子树 → 右子树 → 根节点

5.4.2 二叉树的存储结构

二叉树的存储结构有两种常见方式：**顺序存储**和**链式存储**。

1. 顺序存储结构

顺序存储结构是通过一个数组来存储二叉树。对于树中的每一个节点，我们可以使用数组中的一个元素来表示，数组的下标表示节点的位置。具体规则如下：

- 根节点存储在数组的第一个元素（下标为 0）。
- 对于一个节点，其左子节点的下标是 `2i + 1`，右子节点的下标是 `2i + 2`，父节点的下标是 `(i - 1) / 2`（假设节点从下标 0 开始）。

顺序存储适用于完全二叉树或者节点数固定的二叉树，因为其内存连续分配，查找节点的时间复杂度为 O(1)。

优点：

- 适用于节点数固定且较为均衡的二叉树。
- 存储方式简单，访问节点直接通过数组下标。

缺点：

- 如果树的节点不均匀分布，顺序存储会浪费大量空间。
- 插入和删除操作相对复杂，因为涉及到数组的移动。

2. 链式存储结构

链式存储结构是通过链表节点来存储二叉树。每个节点包含一个数据元素和两个指针，分别指向其左子节点和右子节点。每个二叉树节点的结构如下：

```
struct TreeNode {
    int data;                   // 节点的数据
    TreeNode* left;             // 指向左子节点的指针
    TreeNode* right;            // 指向右子节点的指针

    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}  // 构造函数
};
```

优点：

- 动态分配内存，适用于节点数不固定的情况。
- 适合实现不规则的树结构（例如，满二叉树、完全二叉树之外的二叉树）。

缺点：

- 存储开销较大，每个节点需要额外的空间来存储指针。
- 访问节点时需要通过指针进行查找，时间复杂度为 O(n)。

小结

在本节中，我们详细讨论了二叉树的性质和存储结构。我们介绍了二叉树的基本性质，如节点数、叶子节点数、高度、平衡性等，同时阐述了二叉树的两种常见存储结构——顺序存储和链式存储。每种存储结构有其优缺点，具体的选择取决于应用场景和二叉树的特点。

5.5.1 遍历二叉树

遍历二叉树是访问树中每个节点的过程。二叉树的遍历方式通常有三种：前序遍历、中序遍历和后序遍历。在每种遍历方式中，我们依照不同的顺序访问根节点、左子树和右子树。遍历二叉树不仅有助于理解二叉树的结构，还广泛应用于很多算法和应用程序，如树的排序、搜索等。

1. 前序遍历 (Pre-order Traversal)

前序遍历的顺序是：**根节点** → **左子树** → **右子树**。

2. 中序遍历 (In-order Traversal)

中序遍历的顺序是：**左子树** → **根节点** → **右子树**。

3. 后序遍历 (Post-order Traversal)

后序遍历的顺序是：**左子树** → **右子树** → **根节点**。

在这三种遍历中，每种遍历都能遍历所有节点，但是访问的顺序不同。

5.5.1.1 C++ 代码实现

我们将在 C++ 中实现二叉树的前序、中序、后序遍历，并使用递归方式来进行遍历。

二叉树的结构和遍历实现

```
#include <iostream>
using namespace std;

// 二叉树节点定义
struct BinaryTreeNode {
    int data;                    // 节点的数据
    BinaryTreeNode* left;        // 左子节点
    BinaryTreeNode* right;       // 右子节点

    BinaryTreeNode(int value) : data(value), left(nullptr), right(nullptr) {} // 构造函数
};

// 前序遍历
void preOrderTraversal(BinaryTreeNode* root) {
    if (root == nullptr) return; // 如果节点为空，返回
    cout << root->data << " ";   // 访问根节点
    preOrderTraversal(root->left);  // 递归遍历左子树
    preOrderTraversal(root->right); // 递归遍历右子树
}

// 中序遍历
void inOrderTraversal(BinaryTreeNode* root) {
    if (root == nullptr) return; // 如果节点为空，返回
    inOrderTraversal(root->left);   // 递归遍历左子树
    cout << root->data << " ";      // 访问根节点
    inOrderTraversal(root->right);  // 递归遍历右子树
}

// 后序遍历
void postOrderTraversal(BinaryTreeNode* root) {
    if (root == nullptr) return; // 如果节点为空，返回
    postOrderTraversal(root->left);  // 递归遍历左子树
    postOrderTraversal(root->right); // 递归遍历右子树
    cout << root->data << " ";       // 访问根节点
}

int main() {
    // 创建二叉树
    BinaryTreeNode* root = new BinaryTreeNode(1);
    root->left = new BinaryTreeNode(2);
    root->right = new BinaryTreeNode(3);
    root->left->left = new BinaryTreeNode(4);
    root->left->right = new BinaryTreeNode(5);

    // 打印二叉树的遍历
    cout << "Preorder Traversal: ";
    preOrderTraversal(root);
    cout << endl;

    cout << "Inorder Traversal: ";
    inOrderTraversal(root);
    cout << endl;
```

```
    cout << "Postorder Traversal: ";
    postOrderTraversal(root);
    cout << endl;

    return 0;
}
```

5.5.1.2 代码说明

1. **节点定义**：
   - `BinaryTreeNode` 结构体表示二叉树的节点，其中 `data` 存储节点的值，`left` 和 `right` 分别指向左子节点和右子节点。
2. **前序遍历**：
   - 前序遍历遵循根节点 → 左子树 → 右子树的顺序，递归地访问每个节点。
3. **中序遍历**：
   - 中序遍历遵循左子树 → 根节点 → 右子树的顺序，递归地访问每个节点。
4. **后序遍历**：
   - 后序遍历遵循左子树 → 右子树 → 根节点的顺序，递归地访问每个节点。

5.5.1.3 运行结果

```
Preorder Traversal: 1 2 4 5 3
Inorder Traversal: 4 2 5 1 3
Postorder Traversal: 4 5 2 3 1
```

在上述代码中，创建了一个简单的二叉树结构，并分别实现了三种遍历方式。通过递归实现遍历，代码简单且易于理解。每个遍历方法都会按特定顺序访问树中的节点，并输出节点的数据。

5.5.2 线索二叉树

线索二叉树（Threaded Binary Tree）是二叉树的一种特殊形式，旨在解决普通二叉树中指向空指针的问题。在线索二叉树中，空指针被“线索”所替代，即通过空指针指向二叉树的前驱或后继节点。

1. 线索二叉树的特点

- **空指针替换为线索**：在普通的二叉树中，叶节点的左右指针为空，而在线索二叉树中，这些空指针会指向节点的前驱或者后继节点。
- **前驱指针和后继指针**：线索二叉树中的空左指针指向节点的前驱节点，空右指针指向节点的后继节点。
- **中序线索化**：最常见的线索二叉树是中序线索二叉树，它利用空指针指向节点的中序前驱或后继。

2. 线索二叉树的结构定义

我们首先定义一个线索二叉树节点的结构体。这个结构体包含了一个标志位，表示当前指针是指向子节点还是指向线索。

```
#include <iostream>
using namespace std;

// 线索二叉树节点定义
struct ThreadedBinaryTreeNode {
    int data;                    // 节点的数据
    ThreadedBinaryTreeNode* left;  // 左子节点或前驱节点
    ThreadedBinaryTreeNode* right; // 右子节点或后继节点
    bool leftThread;              // 如果是线索，leftThread 为 true
    bool rightThread;             // 如果是线索，rightThread 为 true

    ThreadedBinaryTreeNode(int value) : data(value), left(nullptr), right(nullptr),
leftThread(false), rightThread(false) {} // 构造函数
};
```

在这个结构中，`leftThread` 和 `rightThread` 是布尔值，用来标记左指针和右指针是否为线索。如果为 `true`，表示该指针不是指向子节点，而是指向前驱或后继节点。

---

3. 中序线索化的实现

为了更好地理解线索二叉树的实现，我们将实现一个简单的中序线索化操作。中序线索化的目标是遍历树并将空指针（如果有）替换为相应的前驱和后继指针。

```
#include <iostream>
using namespace std;

struct ThreadedBinaryTreeNode {
    int data;
    ThreadedBinaryTreeNode* left;
    ThreadedBinaryTreeNode* right;
    bool leftThread;
    bool rightThread;

    ThreadedBinaryTreeNode(int value) : data(value), left(nullptr), right(nullptr),
leftThread(false), rightThread(false) {}
};

// 中序线索化二叉树
void inOrderThread(ThreadedBinaryTreeNode* root) {
    if (!root) return;

    ThreadedBinaryTreeNode* prev = nullptr;  // 用于保存前驱节点
    ThreadedBinaryTreeNode* current = root;

    // 中序遍历
    while (current != nullptr) {
        // 找到最左边的节点
        while (current->left != nullptr && !current->leftThread) {
            current = current->left;
        }
```

```
        // 处理当前节点
        if (prev && current->left == nullptr) {
            current->left = prev;
            current->leftThread = true;
        }

        cout << current->data << " ";

        if (current->right == nullptr || current->rightThread) {
            prev = current;
            current = current->right;
        } else {
            prev = nullptr;
            current = current->right;
        }
    }
}

int main() {
    // 创建节点
    ThreadedBinaryTreeNode* root = new ThreadedBinaryTreeNode(1);
    root->left = new ThreadedBinaryTreeNode(2);
    root->right = new ThreadedBinaryTreeNode(3);
    root->left->left = new ThreadedBinaryTreeNode(4);
    root->left->right =
```

new ThreadedBinaryTreeNode(5);

```
// 中序线索化
cout << "InOrder Traversal (Threaded): ";
inOrderThread(root);
cout << endl;

return 0;
```

}

```
---

#### 4. 代码说明

1. **节点结构**:
   - 结构体 `ThreadedBinaryTreeNode` 包含了 `data`、`left`、`right` 等基本信息，以及 `leftThread` 和 `rightThread` 标志，指示是否为线索。

2. **中序线索化操作**:
   - 在遍历二叉树时，空的左指针会指向前驱节点，空的右指针会指向后继节点。

3. **遍历过程**:
   - 我们采用中序遍历的方式，并在遍历过程中将空指针替换为线索。

---
```

5. 运行结果

```bash
InOrder Traversal (Threaded): 4 2 5 1 3
```

小结

- 在本节中，我们介绍了二叉树的三种基本遍历方式：前序遍历、中序遍历和后序遍历，并实现了对应的 C++ 代码。
- 另外，我们还介绍了线索二叉树的概念，并实现了一个简单的中序线索化过程，展示了如何通过线索来替代空指针，使二叉树的遍历更加高效。线索二叉树可以在遍历时减少对空指针的判断，并加速树的遍历过程。

5.6 树和森林

树和森林是数据结构中非常重要的概念。树是由若干个节点通过边连接而成的非线性数据结构，而森林则是由若干棵树组成的集合。树和森林广泛应用于文件系统、数据库索引、解析树等领域。在这一节中，我们将介绍树的存储结构，森林与二叉树的转换，以及树和森林的遍历方法。

5.6.1 树的存储结构

树的存储结构有两种常见的实现方式：**顺序存储结构**和**链式存储结构**。

1. 顺序存储结构

顺序存储结构是使用一个数组来存储树的节点。在顺序存储结构中，树的节点之间的父子关系通过数组下标来表示。例如，对于一棵树来说：

- 假设树的根节点存储在数组的第一个位置（下标为 0）。
- 对于一个节点，其左子节点的下标为 `2i + 1`，右子节点的下标为 `2i + 2`（其中 `i` 是该节点的数组下标）。
- 父节点的下标为 `(i - 1) / 2`（`i` 为当前节点的数组下标）。

这种存储方式适用于节点数目固定、结构比较规则的树，特别是完全二叉树。

2. 链式存储结构

链式存储结构是通过节点之间的指针来表示树的结构。每个节点包含一个数据域和多个子节点指针。在这种结构中，我们可以为每个节点定义一个**链表**或**链式数组**来存储它的所有子节点。

对于树的链式存储结构，每个节点的定义通常如下：

```
struct TreeNode {
    int data;                   // 节点的数据
    TreeNode* parent;           // 指向父节点的指针
    vector<TreeNode*> children; // 存储子节点的指针集合

    TreeNode(int value) : data(value), parent(nullptr) {}
};
```

这种结构可以处理任意类型的树（不仅限于二叉树），适用于节点数目不固定、树的形态比较灵活的场景。

5.6.2 森林与二叉树的转换

在数据结构中，森林是若干棵互不相交的树的集合。森林与二叉树之间可以通过一种非常特殊的方式进行转换。通常的做法是将森林转化为一棵二叉树，反之亦然。

1. 森林转二叉树

将森林转换为二叉树的基本思想是**通过左子树链表**和**右子树链表**的方式来表示森林。每一棵树的根节点被转换成二叉树的一棵子树。

- **左子树**：指向当前节点的第一个子节点。
- **右子树**：指向当前节点的右兄弟（即同一父节点的下一个兄弟节点）。

假设森林由树 `T1, T2, ..., Tn` 构成，每一棵树 `Ti` 的根节点变成二叉树的根节点，并且树 `Ti` 的左子树指向其第一个子节点，右子树指向其右兄弟。

2. 二叉树转森林

将二叉树转换为森林的基本思想是将每个二叉树的左子树和右子树分别分解为独立的树，并把左子树和右子树分别作为森林中的元素。根节点作为森林的“父节点”，其左子树变成森林的一棵树，右子树变成森林的另一棵树。

例如，二叉树的左子树和右子树本身都可以是一个独立的树，这样转换后就得到了一个森林。

5.6.3 树和森林的遍历

树和森林的遍历方式与二叉树的遍历类似，但是有一些额外的细节需要注意。由于森林是多个树的集合，因此遍历森林时，我们需要分别遍历每棵树。

1. 树的遍历

树的遍历方式通常有**前序遍历**、**中序遍历**和**后序遍历**，这些遍历方式和二叉树的遍历方式类似。唯一不同的是，树的节点可以有多个子节点，因此我们需要遍历每个节点的所有子节点。

以下是树的前序遍历的基本思路：

- 前序遍历

  （根节点 → 左子树 → 右子树）：

  - 访问根节点。
  - 遍历左子树。
  - 遍历右子树。

```
void preOrderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->data << " "; // 访问根节点
    for (TreeNode* child : root->children) {
        preOrderTraversal(child); // 遍历每个子节点
    }
}
```

2. 森林的遍历

森林的遍历实际上是对森林中每一棵树的遍历。我们可以通过遍历每一棵树来实现森林的遍历。假设森林包含若干棵树 `T1, T2, ..., Tn`，遍历森林的操作就是遍历每棵树 `Ti`。

```
void traverseForest(vector<TreeNode*> forest) {
    for (TreeNode* tree : forest) {
        preOrderTraversal(tree); // 对每棵树进行遍历
        cout << endl;  // 每棵树遍历后换行
    }
}
```

在上面的代码中，我们首先定义了一个 `preOrderTraversal` 函数来遍历单棵树，然后通过 `traverseForest` 函数遍历森林中所有的树。

小结

在这一节中，我们讨论了树和森林的相关知识。主要内容包括：

- **树的存储结构**：介绍了树的顺序存储和链式存储结构，强调了它们的优缺点。
- **森林与二叉树的转换**：讲解了如何将森林转换为二叉树，反之亦然。
- **树和森林的遍历**：我们介绍了树的前序遍历，并扩展到森林的遍历，即遍历森林中的每一棵树。

5.7 哈夫曼树及其应用

哈夫曼树（Huffman Tree）是一种用于数据压缩的最优二叉树。它由哈夫曼编码算法构造，广泛应用于文件压缩、图像压缩以及视频编码等领域。哈夫曼树的构造利用了频率或权重的思想，在编码过程中通过赋予频率较高的字符较短的编码，频率较低的字符较长的编码，从而实现数据的有效压缩。

5.7.1 哈夫曼树的基本概念

哈夫曼树是由 **贪心算法** 构建的二叉树，其基本思想是：在编码时给频率高的字符分配较短的编码，频率低的字符分配较长的编码，以此减少整体数据的存储量。

哈夫曼树的构造过程涉及到以下几个基本概念：

- **字符的频率**：每个字符在文本中出现的频率。
- **哈夫曼树的性质**：哈夫曼树是一棵带权路径长度最小的二叉树，其中每个叶子节点表示一个字符，而每条边的权值通常与字符的出现频率成正比。

哈夫曼树具有以下特点：

1. **完全二叉树**：每个内部节点都有两个子节点。
2. **最小带权路径长度**：该树的带权路径长度（即字符的编码长度的加权和）最小，具有最优性。

5.7.2 哈夫曼树的构造算法

构造哈夫曼树的算法是基于贪心算法的。它的基本思想是：

1. **初始化**：把每个字符作为一个叶子节点，每个节点的权值是该字符出现的频率。
2. **选择最小的两个节点**：从森林中选择频率最小的两个节点，合并为一个新的节点，该节点的权值为两个最小节点的权值之和。

3. **构建哈夫曼树**：重复步骤 2，直到只剩下一个节点，即哈夫曼树的根节点。

具体步骤如下：

1. **初始化**：构建一个最小堆（或优先队列），将所有节点按频率从小到大排序。
2. **合并节点**：从堆中取出两个最小的节点，构建一个新的节点，并将这个新节点的频率设置为两个节点的频率之和，然后将新节点插入堆中。
3. **重复合并**：重复这个过程，直到堆中只剩下一个节点，此时的节点即为哈夫曼树的根节点。

C++ 代码实现

以下是基于 C++ 的哈夫曼树构造算法的实现：

```
#include <iostream>
#include <queue>
#include <vector>
#include <unordered_map>
using namespace std;

// 哈夫曼树节点
struct HuffmanNode {
    char ch;            // 字符
    int freq;           // 字符频率
    HuffmanNode* left;  // 左子节点
    HuffmanNode* right; // 右子节点

    // 构造函数
    HuffmanNode(char c, int f) : ch(c), freq(f), left(nullptr), right(nullptr) {}
};

// 自定义比较器，用于优先队列
struct Compare {
    bool operator()(HuffmanNode* a, HuffmanNode* b) {
        return a->freq > b->freq;  // 最小堆，频率小的优先
    }
};

// 构造哈夫曼树
HuffmanNode* buildHuffmanTree(const unordered_map<char, int>& freqMap) {
    priority_queue<HuffmanNode*, vector<HuffmanNode*>, Compare> pq;

    // 将所有字符及其频率放入优先队列
    for (auto& entry : freqMap) {
        pq.push(new HuffmanNode(entry.first, entry.second));
    }

    // 合并节点直到队列中只剩下一个节点
    while (pq.size() > 1) {
        // 取出两个最小频率的节点
        HuffmanNode* left = pq.top(); pq.pop();
        HuffmanNode* right = pq.top(); pq.pop();

        // 创建新的父节点
        HuffmanNode* parent = new HuffmanNode('\0', left->freq + right->freq);  // 不存储字符
```

```
        parent->left = left;
        parent->right = right;

        // 将新的节点插入优先队列
        pq.push(parent);
    }

    // 返回哈夫曼树的根节点
    return pq.top();
}

// 递归遍历哈夫曼树并生成哈夫曼编码
void generateHuffmanCodes(HuffmanNode* root, string code, unordered_map<char, string>&
huffmanCodes) {
    if (root == nullptr) return;

    // 叶子节点，保存字符和对应的编码
    if (root->ch != '\0') {
        huffmanCodes[root->ch] = code;
    }

    // 递归遍历左右子树
    generateHuffmanCodes(root->left, code + "0", huffmanCodes);
    generateHuffmanCodes(root->right, code + "1", huffmanCodes);
}

int main() {
    // 示例：字符及其频率
    unordered_map<char, int> freqMap = {
        {'a', 5},
        {'b', 9},
        {'c', 12},
        {'d', 13},
        {'e', 16},
        {'f', 45}
    };

    // 构造哈夫曼树
    HuffmanNode* root = buildHuffmanTree(freqMap);

    // 获取哈夫曼编码
    unordered_map<char, string> huffmanCodes;
    generateHuffmanCodes(root, "", huffmanCodes);

    // 输出哈夫曼编码
    cout << "Huffman Codes:" << endl;
    for (auto& entry : huffmanCodes) {
        cout << entry.first << ": " << entry.second << endl;
    }

    return 0;
}
```

5.7.3 哈夫曼编码

哈夫曼编码是哈夫曼树的应用之一。哈夫曼编码通过将频率高的字符编码为较短的二进制码，频率低的字符编码为较长的二进制码，来达到数据压缩的目的。

在哈夫曼树中，路径从根节点到叶子节点的每个分支决定了该叶子节点（即字符）的哈夫曼编码：

- 向左分支表示编码为 `0`。
- 向右分支表示编码为 `1`。

哈夫曼编码的主要优势是**前缀编码**（即没有编码是另一个编码的前缀）。这种特性保证了编码的唯一性和无歧义性。

示例

假设有以下字符及其频率：

```
a: 5
b: 9
c: 12
d: 13
e: 16
f: 45
```

通过哈夫曼树构建和编码的过程，我们得到如下的哈夫曼编码：

```
a: 1100
b: 1101
c: 111
d: 10
e: 0
f: 1
```

根据上述编码，可以看到字符 `f`（频率最高）拥有最短的编码，而字符 `a`（频率较低）则拥有较长的编码。

小结

在这一节中，我们学习了哈夫曼树及其应用的几个重要方面：

1. **哈夫曼树的基本概念**：哈夫曼树是一种带有最小带权路径长度的二叉树，常用于数据压缩。
2. **哈夫曼树的构造算法**：通过贪心算法，我们将频率最低的节点合并，逐步构造哈夫曼树。
3. **哈夫曼编码**：通过遍历哈夫曼树的路径，我们可以为每个字符生成一个唯一的二进制编码，用于数据压缩。

哈夫曼编码不仅具有最优性，而且由于其简单的结构和编码方式，广泛应用于文件压缩算法（如 ZIP 和 GZIP）以及图像压缩算法（如 JPEG 和 PNG）。

第 6 章 图

图是一种广泛应用于计算机科学和其他领域的重要数据结构，它由若干个节点（也称为顶点）和连接这些节点的边组成。图在许多实际问题中都有应用，如社交网络、计算机网络、路径规划、推荐系统等。本章将介绍图的定义、基本术语以及一些常见的图的应用。

6.1 图的定义和基本术语

图（Graph）是由一组顶点（V）和一组边（E）组成的集合。每条边都连接图中的两个顶点，图的边可以是有向的，也可以是无向的，顶点和边的排列组合定义了不同类型的图。

6.1.1 图的定义

图 G 可以表示为一个有序二元组：

```
G = (V, E)
```

其中：

- **V**：是图中的顶点集合，V = {v_1, v_2, ..., vn}，表示图中所有的节点。
- **E**：是图中的边集合，E = {(v_1, v_2), (v_2, v_3), ...}，每条边连接图中的两个顶点。

图的种类：

- **有向图（Directed Graph）**：图中的边有方向，表示从一个顶点到另一个顶点的关系。每条边用有序对表示 (v_1, v_2)，表示从顶点 v_1 到顶点 v_2。
- **无向图（Undirected Graph）**：图中的边没有方向，表示两个顶点之间的关系。每条边用无序对表示 $\{v_1, v_2\}$，表示顶点 v_1 和顶点 v_2 之间的关系。

此外，还有一些特殊的图：

- **加权图（Weighted Graph）**：图中的每条边都有一个权值（或称为成本、长度等）。
- **无权图（Unweighted Graph）**：图中的每条边没有权值。
- **完全图（Complete Graph）**：图中的每对不同的顶点都有一条边。
- **连通图（Connected Graph）**：无向图中任意两个顶点都有路径相连。

6.1.2 图的基本术语

图的基本术语包括以下几个方面：

1. **顶点（Vertex）**：图中的每一个节点称为顶点（有时也称为“点”）。
2. **边（Edge）**：连接两个顶点的线段称为边。对于无向图，边是无方向的；对于有向图，边是有方向的。
3. 度（Degree）：
   - **顶点的度（Degree of a Vertex）**：一个顶点的度是与该顶点相连接的边的数量。在无向图中，顶点的度是该顶点的边的数量；在有向图中，顶点的度可以分为**入度**（指向该顶点的边的数量）和**出度**（从该顶点指向其他顶点的边的数量）。
4. **路径（Path）**：从一个顶点到另一个顶点的顶点序列，序列中的每两个连续顶点之间都有边连接。
5. **简单路径（Simple Path）**：路径中的每个顶点都不重复。
6. **回路（Cycle）**：从某一顶点出发，沿着边回到该顶点的路径。若路径中的所有顶点都不重复，则称为**简单回路**。
7. 连通性：
   - **连通图（Connected Graph）**：无向图中的任意两个顶点都可以通过路径相连。
   - **强连通图（Strongly Connected Graph）**：有向图中的任意两个顶点都有路径可以互相到达。

6.2 案例引入

在实际应用中，图有很多重要的应用场景。我们来看几个典型的应用案例。

1. 社交网络

社交网络中的用户可以看作图的顶点，用户之间的关系（如“朋友”关系）可以看作边。社交网络中的一些问题，如“最短路径”问题（两个人之间的最短友谊链）、“推荐系统”问题（通过朋友推荐朋友）等，都可以通过图来建模和解决。

2. 路径规划

在地图上，城市可以看作图的顶点，城市之间的道路可以看作边。每条道路可能有不同的长度或交通状况，可以在图中赋予不同的权值。路径规划问题（例如找到从一个城市到另一个城市的最短路径）可以通过图算法来解决，如Dijkstra算法、A*算法等。

3. 计算机网络

计算机网络是由许多计算机和它们之间的连接（例如路由器、交换机）组成的图。在计算机网络中，节点代表计算机，边代表计算机之间的通信链路。网络中最短路径、最小生成树等问题可以通过图算法来求解。

4. Web页面链接

Web页面可以看作图中的顶点，而页面之间的超链接可以看作边。搜索引擎会根据网页之间的链接关系来评估页面的相关性和权重，Google的PageRank算法就是基于图的思想，通过计算每个页面的权重来判断页面的重要性。

5. 推荐系统

在推荐系统中，图可以用来表示用户与物品的关系。用户和物品都是图中的顶点，用户与物品之间的关系（如购买、评分）可以用边来表示。基于图的推荐算法，如协同过滤算法，能够为用户推荐他们可能喜欢的物品。

这些应用案例表明了图在不同领域中的广泛应用，图的相关算法为解决这些问题提供了理论基础和技术支持。

小结

本节内容主要介绍了图的定义和基本术语，包括：

- **图的定义**：图是由顶点和边组成的集合，包含有向图、无向图、加权图等多种形式。
- **基本术语**：如顶点、边、度、路径、回路等。
- **应用案例**：通过社交网络、路径规划、计算机网络、Web页面链接、推荐系统等案例，展示了图在实际中的重要应用。

6.3 图的类型定义

在图的理论中，图的类型定义主要是根据图的边的特性和结构来分类的。图的类型不仅影响图的存储方式，还影响我们选择适当的图算法来解决具体的问题。常见的图的分类包括有向图和无向图、加权图和无权图、连通图和非连通图等。本节将详细介绍这些常见图的类型及其定义。

6.3.1 无向图（Undirected Graph）

无向图是一种图，其中的边没有方向，即边连接两个顶点的关系是对称的。例如，在社交网络中，朋友关系是对称的，因此用无向图表示更为合适。

- **定义**：无向图 G=(V,E)G = (V, E)，其中：
  - VV 是顶点集合，
  - EE 是边的集合，每条边 ee 都是顶点对 (vi,vj)(v_i, v_j) 的无序对。
- **性质**：无向图的边没有方向，意味着边 (vi,vj)(v_i, v_j) 和 (vj,vi)(v_j, v_i) 是等价的。
- **例子**：社交网络中的朋友关系。

无向图的表示：

对于无向图的存储，可以使用邻接矩阵、邻接表等数据结构。无向图的邻接矩阵是对称的，因为边的方向没有区分。

6.3.2 有向图（Directed Graph）

有向图是一种图，其中的边是有方向的，即每条边都有一个起点和一个终点。边的方向从一个顶点指向另一个顶点，因此有向图是一个有序的顶点对。

- **定义**：有向图 G=(V,E)G = (V, E)，其中：
  - VV 是顶点集合，
  - EE 是边的集合，每条边 ee 都是顶点对 (vi,vj)(v_i, v_j) 的有序对。
- **性质**：有向图中的边是有方向的，意味着 (vi,vj)(v_i, v_j) 和 (vj,vi)(v_j, v_i) 是不同的边。
- **例子**：计算机网络中的数据包传输，网页之间的超链接，交通流向等。

有向图的表示：

有向图的邻接矩阵是非对称的，因为边有方向，因此 (vi,vj)(v_i, v_j) 与 (vj,vi)(v_j, v_i) 不一定相等。

6.3.3 加权图（Weighted Graph）

加权图是一种图，其中每条边都有一个与之相关的权值（或称为成本、长度）。在加权图中，边的权值通常表示边的某种特性，如距离、费用、时间等。

- **定义**：加权图 G=(V,E)G = (V, E)，其中：
  - VV 是顶点集合，
  - EE 是边的集合，每条边 e=(vi,vj)e = (v_i, v_j) 都关联一个权值 wijw_{ij}。
- **性质**：加权图中的边不仅包含连接两个顶点的关系，还包含边的权值。权值可以是任意的数值，可以表示距离、费用、容量等。
- **例子**：地图上的道路网络，边的权值可以表示距离或通行时间；计算机网络中，边的权值可以表示带宽。

加权图的表示：

加权图通常使用邻接矩阵或邻接表来表示，每条边都有一个权值。如果边没有权值，则可以看作是一个无权图。

6.3.4 无权图（Unweighted Graph）

无权图是一种图，其中每条边都没有权值。无权图只关心顶点之间是否有连接关系，而不考虑连接的强度或其他性质。

- **定义**：无权图 G=(V,E)G = (V, E)，其中：
  - VV 是顶点集合，
  - EE 是边的集合，每条边仅表示两个顶点之间的连接关系，而没有权值。
- **性质**：无权图的边没有权值，边的存在性即表示连接关系，通常用于表示存在性问题（例如“是否有路径连接两个节点”）。
- **例子**：朋友关系网络（只关心是否为朋友，不关心朋友关系的强度）。

无权图的表示：

无权图的邻接矩阵和邻接表表示都只关心边的存在与否，边可以用0和1表示，0表示无连接，1表示有连接。

6.3.5 完全图（Complete Graph）

完全图是一种图，其中每对不同的顶点都有一条边直接连接。对于一个顶点集合中的每两个不同的顶点 viv_i 和 vjv_j，都有一条边连接它们。

- **定义**：完全图是一个无向图 G=(V,E)G = (V, E)，其中每对不同的顶点 viv_i 和 vjv_j 都有一条边 (vi,vj)∈E(v_i, v_j) \in E。
- **性质**：一个包含 nn 个顶点的完全图有 n(n−1)2\frac{n(n-1)}{2} 条边。
- **例子**：在社交网络中，如果每个用户都与所有其他用户直接有连接关系，那么这个网络可以用完全图表示。

完全图的表示：

完全图的邻接矩阵中，除了对角线上的元素（表示顶点自己与自己没有边）外，其他所有元素都为1，表示每两个不同的顶点之间都有边。

6.3.6 连通图（Connected Graph）

连通图是一种无向图，其中任意两个顶点之间都有一条路径相连。换句话说，连通图中的任意两个顶点都可以通过边达到。

- **定义**：无向图 G=(V,E)G = (V, E) 是连通的，当且仅当图中的任意两个不同的顶点 viv_i 和 vjv_j 都存在一条路径连接它们。
- **性质**：连通图中的所有顶点都是相互可达的。如果图不是连通的，则称为**非连通图**。
- **例子**：城市之间的道路网络。如果每个城市都可以通过一系列道路互相到达，则该图是连通的。

连通图的表示：

连通图的邻接矩阵表示中，任意两个顶点之间都有路径连接。非连通图则可能存在一些顶点是孤立的。

6.3.7 强连通图（Strongly Connected Graph）

强连通图是一个有向图，在这个图中，任意两个顶点都可以互相到达，即从顶点 viv_i 可以通过有向边到达顶点 vjv_j，同时也可以从 vjv_j 回到 viv_i。

- **定义**：有向图 G=(V,E)G = (V, E) 是强连通的，当且仅当任意两个不同的顶点 viv_i 和 vjv_j 都有路径可以互相到达。

- **性质**：强连通图中的每两个顶点都有双向的路径连接。如果有向图不是强连通的，则称其为**非强连通图**。
- **例子**：一个社交网络中的双向互动关系，如果两个用户之间可以互相评论和回复，那么它们之间是强连通的。

强连通图的表示：

强连通图的邻接矩阵没有任何元素是零，表示每个顶点都可以通过路径到达其他顶点。

小结

本节介绍了图的不同类型及其定义，包括无向图、有向图、加权图、无权图、完全图、连通图和强连通图等。这些类型的不同决定了图的存储方式和适用的算法。了解这些图的类型可以帮助我们根据问题的实际需求选择合适的图模型，并设计高效的图算法。

6.4 图的存储结构

图的存储结构主要有四种常见的表示方式：邻接矩阵、邻接表、十字链表和邻接多重表。每种存储结构各有优缺点，适用于不同的应用场景。选择合适的存储结构可以有效地提高图算法的性能。

6.4.1 邻接矩阵（Adjacency Matrix）

邻接矩阵是一种二维数组，用来表示图中顶点之间的连接关系。对于一个有 nn 个顶点的图，邻接矩阵是一个 n×nn \times n 的矩阵，其中每个元素表示顶点之间是否有边连接。

- **定义**：
  - 对于无向图，邻接矩阵是对称的，即 A[i][j]=A[j][i]A[i][j] = A[j][i]。
  - 对于有向图，矩阵元素表示从顶点 ii 到顶点 jj 是否有边。
- **矩阵表示**：
  - 如果图是无权图，矩阵元素为 1 表示有边，0 表示无边。
  - 如果图是加权图，矩阵元素为边的权值，若没有边，则为 0 或某个特殊值（如无穷大）。
- **优缺点**：
  - 优点：
    - 适用于稠密图（边数较多的图）。
    - 可以在 O(1)O(1) 时间内判断任意两个顶点之间是否有边。
  - 缺点：
    - 空间复杂度较高，对于稀疏图（边数较少的图），存储空间浪费较多。
    - 不适合动态变化的图（如频繁增删边）。
- **邻接矩阵示例**：对于以下无向图：

```
0 -- 1
|    |
2 -- 3
```

邻接矩阵表示为：

```
A = [
    [0, 1, 1, 0],
    [1, 0, 1, 0],
    [1, 1, 0, 1],
    [0, 0, 1, 0]
]
```

这里，顶点 00 和顶点 11 之间有一条边，因此 A[0][1]=1A[0][1] = 1，而 A[0][3]=0A[0][3] = 0，表示顶点 00 和顶点 33 没有边。

6.4.2 邻接表（Adjacency List）

邻接表是另一种常见的图存储结构。它通过为图中的每个顶点维护一个链表（或其他线性结构）来表示与该顶点相邻的所有顶点。

- **定义**：
  - 对于每个顶点 viv_i，维护一个链表，链表中的元素表示与 viv_i 相连接的其他顶点。
- **优缺点**：
  - 优点：
    - 节省空间，特别适用于稀疏图。
    - 对于稀疏图，边的存储量比邻接矩阵节省很多空间。
    - 可以有效地存储动态变化的图（如增删边）。
  - 缺点：
    - 在查询两个顶点是否有边连接时，需要遍历链表，时间复杂度为 O(k)O(k)，其中 kk 为顶点 viv_i 的度数。
- **邻接表示例**：对于以下无向图：

```
0 -- 1
|    |
2 -- 3
```

邻接表表示为：

```
0: 1 -> 2
1: 0 -> 3
2: 0 -> 3
3: 1 -> 2
```

其中，顶点 0 的邻接表存储了顶点 1 和 2，表示顶点 0 与这两个顶点有边连接。

6.4.3 十字链表（Orthogonal List）

十字链表是图的另一种存储结构，专门为有向图设计。它结合了邻接矩阵和邻接表的优点，能够有效存储有向图的顶点和边。十字链表为每个边设置了两个指针，一个指向起点，另一个指向终点。十字链表的每个结点通常包含四个指针：一个指向起点的邻接表，一个指向终点的邻接表，以及起点和终点的“逆”链表。

- **定义**：
  - 十字链表使用四种指针来维护信息：
    - **头指针**：指向图的顶点。
    - **出边指针**：指向从该顶点出发的边。
    - **入边指针**：指向指向该顶点的边。
    - **逆边指针**：指向与当前边反向的边。
- **优缺点**：
  - 优点：
    - 十字链表适用于有向图，能够高效地存储有向边。
    - 在存储有向图时，查询顶点的入度和出度都比较方便。
  - 缺点：
    - 存储结构较为复杂，空间开销较大。
    - 对于无向图或稀疏图，使用十字链表的优势不明显。
- **十字链表示例**：假设有一个有向图 GG：

```
0 → 1
↑   ↓
2 ← 3
```

其十字链表表示可能包括以下四个链表：

- 顶点链表：

```
0 → 1 → 2 → 3
```

- 出边链表（从顶点出发的边）：

```
0 → 1
1 → 3
2 → 0
3 → 2
```

- 入边链表（指向顶点的边）：

```
0 → 2
1 → 0
2 → 3
3 → 1
```

- 逆边链表

  :

```
0 ← 2
1 ← 0
2 ← 3
3 ← 1
```

6.4.4 邻接多重表（Adjacency Multilist）

邻接多重表是一种图的存储结构，适用于包含多重边（即两个顶点之间存在多条边）的图。与邻接表类似，邻接多重表为每个顶点维护一个链表来存储与该顶点相连的顶点信息，但它允许一个顶点有多条边连接到另一个顶点，因此每个链表中可以有多个相同的边。

- **定义**：
  - 每个顶点对应一个邻接表，其中每个节点表示与该顶点相连的其他顶点。
  - 允许在邻接表中存储多条边，适用于有多重边的图。
- **优缺点**：
  - 优点

    :

    - 支持多重边，适用于需要存储重复连接关系的图。
  - 缺点

    :

    - 存储空间较大。
    - 对于没有多重边的图，使用邻接多重表可能浪费空间。
- **邻接多重表示例**：对于以下图，两个顶点 0 和 1 之间存在两条边：

```
0 -- 1
|    |
2 -- 3
```

邻接多重表可以表示为：

```
0: 1 → 1 → 2
1: 0 → 3
2: 0 → 3
3: 1 → 2
```

其中，顶点 0 和 1 之间有两条边，因此在邻接表中会出现两个 `1`。

小结

本节介绍了图的四种常见存储结构：

- **邻接矩阵**：适用于稠密图，查询快速，但存储空间较大。
- **邻接表**：适用于稀疏图，存储节省空间，但查询顶点连接关系时可能较慢。
- **十字链表**：专门为有向图设计，支持高效的有向边存储和查询，但存储复杂。
- **邻接多重表**：适用于多重边的图，支持存储重复的边，空间开销较大。

6.5 图的遍历

图的遍历是图论中的一种基本操作，主要有两种经典的遍历方法：**深度优先搜索（DFS）和广度优先搜索（BFS）**。这两种遍历方法分别适用于不同的应用场景，能够帮助我们解决各种图相关的问题，例如查找路径、寻找连通组件、拓扑排序等。

6.5.1 深度优先搜索（DFS）

深度优先搜索（DFS）是一种图的遍历算法，它通过从一个起始顶点出发，沿着图的深度方向搜索，直到到达没有未访问的邻居的顶点为止，然后回溯并继续搜索其他路径。DFS 通常使用栈来辅助实现，能够深入到图的每一个分支。

- **基本思想**：
  1. 从起始顶点出发，访问该顶点。
  2. 对于当前顶点的每个未访问的邻居，递归地进行深度优先遍历。
  3. 如果所有邻居都已访问，回溯到上一个顶点，继续遍历其未访问的邻居。
- **实现方式**：
  - **递归实现**：使用系统栈来存储每个递归调用。
  - **非递归实现**：使用显式栈来模拟递归调用的过程。
- **时间复杂度**：
  - DFS 的时间复杂度为 O(V+E)O(V + E)，其中 VV 是顶点数，EE 是边数。
- **适用场景**：
  - 深度优先搜索适用于需要深入探索每个分支的场景，例如寻找路径、拓扑排序、判断图的连通性等。
- **C++ 代码示例（递归实现）**：

```
#include <iostream>
#include <vector>
using namespace std;

// 图的邻接表表示
class Graph {
public:
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

    Graph(int V) {
        this->V = V;
```

```
        adj.resize(V);
    }

    // 添加边
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    // 深度优先搜索
    void DFS(int v, vector<bool>& visited) {
        // 标记当前顶点为已访问
        visited[v] = true;
        cout << v << " "; // 输出当前顶点

        // 递归访问所有邻接的未访问顶点
        for (int neighbor : adj[v]) {
            if (!visited[neighbor]) {
                DFS(neighbor, visited);
            }
        }
    }
};

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 3);

    vector<bool> visited(4, false); // 用于标记顶点是否已访问
    cout << "深度优先搜索结果: ";
    g.DFS(0, visited); // 从顶点0开始DFS
    return 0;
}
```


输出:

```
深度优先搜索结果: 0 1 3 2
```

6.5.2 广度优先搜索（BFS）

广度优先搜索（BFS）是一种图的遍历算法，它通过从一个起始顶点出发，先访问该顶点的所有邻居，然后再依次访问每个邻居的邻居，逐层向外扩展。BFS 通常使用队列来辅助实现，能够按照图的层次结构逐层遍历所有顶点。

- **基本思想**:
  1. 从起始顶点出发，访问该顶点并将其加入队列。
  2. 从队列中取出顶点，访问其所有未访问的邻居，并将这些邻居加入队列。
  3. 重复上述步骤，直到队列为空。
- **实现方式**:

- 使用队列来存储待访问的顶点，按照层次顺序逐个访问。

- **时间复杂度**：
  - BFS 的时间复杂度为 O(V+E)O(V + E)，其中 VV 是顶点数，EE 是边数。
- **适用场景**：
  - 广度优先搜索适用于需要层次遍历的场景，例如最短路径问题、连通分量的查找等。
- **C++ 代码示例**：

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

// 图的邻接表表示
class Graph {
public:
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表

    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }

    // 添加边
    void addEdge(int u, int v) {
        adj[u].push_back(v);
    }

    // 广度优先搜索
    void BFS(int start) {
        vector<bool> visited(V, false); // 标记顶点是否已访问
        queue<int> q; // 用于BFS的队列

        // 标记起始顶点并加入队列
        visited[start] = true;
        q.push(start);

        while (!q.empty()) {
            int v = q.front(); // 取队列头部元素
            q.pop();
            cout << v << " "; // 输出当前顶点

            // 访问所有未访问的邻接顶点
            for (int neighbor : adj[v]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    q.push(neighbor);
                }
            }
        }
    }
```

```
};

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 3);
    g.addEdge(2, 3);

    cout << "广度优先搜索结果: ";
    g.BFS(0); // 从顶点0开始BFS
    return 0;
}
```

输出:

```
广度优先搜索结果: 0 1 2 3
```

小结

1. **深度优先搜索（DFS）**：
   - 通过递归或栈进行图的深度遍历。
   - 时间复杂度：O(V+E)O(V + E)。
   - 适合用于路径查找、拓扑排序、连通性检测等问题。
2. **广度优先搜索（BFS）**：
   - 通过队列进行图的层次遍历。
   - 时间复杂度：O(V+E)O(V + E)。
   - 适合用于最短路径问题、层次遍历、连通分量检测等问题。

6.6 图的应用

图作为一种重要的数据结构，在许多实际问题中得到了广泛应用。通过图的各种算法，能够解决诸如网络优化、路径搜索、排序等问题。以下是图的四个经典应用：最小生成树、最短路径、拓扑排序和关键路径。

6.6.1 最小生成树（Minimum Spanning Tree）

最小生成树（MST）是图论中的一个重要概念，指的是在一个连通加权图中，选择一部分边使得图中的所有顶点都能够连通，且边的总权重最小。最小生成树可以用来解决网络设计问题，例如构建最小成本的网络连接。

- **定义**：
  - **最小生成树**是一个包含图中所有顶点的生成树，且生成树的边的权重之和最小。
- **常用算法**：
  - **Kruskal 算法**：基于贪心策略，通过按权重升序排列边，逐步选择边来构建生成树。
  - **Prim 算法**：从一个起始顶点出发，逐步选择与树相连接的最小边来扩展生成树。
- **应用**：
  - 最小生成树可以用于网络布线、城市的道路规划、电子电路设计等场景。

- **C++ 代码示例（Kruskal 算法）**：

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// 边的结构体
struct Edge {
    int u, v, weight;
    bool operator<(const Edge &e) {
        return weight < e.weight;
    }
};

class Graph {
public:
    int V; // 顶点数
    vector<Edge> edges; // 边的集合

    Graph(int V) {
        this->V = V;
    }

    // 添加边
    void addEdge(int u, int v, int weight) {
        edges.push_back({u, v, weight});
    }

    // 查找祖先
    int find(int parent[], int i) {
        if (parent[i] == -1)
            return i;
        return find(parent, parent[i]);
    }

    // 合并两个集合
    void Union(int parent[], int x, int y) {
        int xroot = find(parent, x);
        int yroot = find(parent, y);
        parent[xroot] = yroot;
    }

    // Kruskal 算法
    void kruskal() {
        sort(edges.begin(), edges.end()); // 按权重排序边

        int parent[V];
        fill(parent, parent + V, -1); // 初始化并查集

        vector<Edge> mst; // 最小生成树

        for (auto &edge : edges) {
```

```
            int x = find(parent, edge.u);
            int y = find(parent, edge.v);

            // 如果不在同一个集合中，加入最小生成树
            if (x != y) {
                mst.push_back(edge);
                Union(parent, x, y);
            }
        }

        // 输出最小生成树的边
        cout << "最小生成树的边: " << endl;
        for (auto &edge : mst) {
            cout << edge.u << " - " << edge.v << " : " << edge.weight << endl;
        }
    }
};

int main() {
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(0, 2, 6);
    g.addEdge(0, 3, 5);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);

    g.kruskal(); // 计算最小生成树
    return 0;
}
```

输出：

```
最小生成树的边:
2 - 3 : 4
0 - 3 : 5
0 - 1 : 10
```

6.6.2 最短路径（Shortest Path）

最短路径问题是图论中的一个经典问题，要求在图中找到从一个起始顶点到所有其他顶点的最短路径。最短路径算法广泛应用于地图导航、网络数据传输、路径规划等领域。

- **常用算法**：
  - **Dijkstra 算法**：适用于没有负权边的图，通过贪心策略逐步确定起点到其他顶点的最短路径。
  - **Bellman-Ford 算法**：适用于带有负权边的图，能够处理负权环路。
  - **Floyd-Warshall 算法**：适用于计算图中所有顶点对之间的最短路径。
- **应用**：
  - 网络路由、导航系统、地铁线路规划等。
- **C++ 代码示例（Dijkstra 算法）**：

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

class Graph {
public:
    int V; // 顶点数
    vector<vector<pair<int, int>>> adj; // 邻接表，存储边的权重

    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }

    // 添加边
    void addEdge(int u, int v, int weight) {
        adj[u].push_back({v, weight});
        adj[v].push_back({u, weight}); // 如果是无向图
    }

    // Dijkstra 算法计算最短路径
    void dijkstra(int start) {
        vector<int> dist(V, INT_MAX); // 存储最短距离
        dist[start] = 0;
        priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
// 优先队列
        pq.push({0, start}); // 起始点入队

        while (!pq.empty()) {
            int u = pq.top().second;
            int d = pq.top().first;
            pq.pop();

            // 遍历当前顶点的所有邻接点
            for (auto &neighbor : adj[u]) {
                int v = neighbor.first;
                int weight = neighbor.second;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.push({dist[v], v});
                }
            }
        }

        // 输出最短路径
        for (int i = 0; i < V; ++i) {
            cout << "从顶点 " << start << " 到顶点 " << i << " 的最短路径是: " << dist[i] <<
endl;
        }
    }
```

```
};

int main() {
    Graph g(5);
    g.addEdge(0, 1, 10);
    g.addEdge(0, 4, 5);
    g.addEdge(1, 2, 1);
    g.addEdge(1, 4, 2);
    g.addEdge(2, 3, 4);
    g.addEdge(3, 0, 7);
    g.addEdge(4, 2, 9);
    g.addEdge(4, 3, 2);

    g.dijkstra(0); // 从顶点0开始计算最短路径
    return 0;
}
```

输出:

```
从顶点 0 到顶点 0 的最短路径是: 0
从顶点 0 到顶点 1 的最短路径是: 8
从顶点 0 到顶点 2 的最短路径是: 9
从顶点 0 到顶点 3 的最短路径是: 7
从顶点 0 到顶点 4 的最短路径是: 5
```

6.6.3 拓扑排序（Topological Sort）

拓扑排序是对有向无环图（DAG）中的顶点进行排序，使得对于图中的每一条有向边 (u,v)(u, v)，顶点 uu 都排在顶点 vv 之前。拓扑排序广泛应用于任务调度、课程安排、项目管理等场景。

- **基本思想**:
  - 拓扑排序可以通过 **深度优先搜索（DFS）** 或 **入度为零的顶点** 的算法实现。
- **应用**:
  - 项目调度、课程安排、编译器中的依赖关系处理等。
- **C++ 代码示例（拓扑排序，Kahn 算法）**:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

class Graph {
public:
    int V; // 顶点数
    vector<vector<int>> adj; // 邻接表
    vector<int> indegree; // 入度

    Graph(int V) {

        this->V = V;
```

```
        adj.resize(V);
        indegree.resize(V, 0
        )
;}
  // 添加边
  void addEdge(int u, int v) {
      adj[u].push_back(v);
      indegree[v]++;
  }

  // 拓扑排序
  void topologicalSort() {
      queue<int> q;

      // 将所有入度为0的顶点入队
      for (int i = 0; i < V; ++i) {
          if (indegree[i] == 0) {
              q.push(i);
          }
      }

      while (!q.empty()) {
          int u = q.front();
          q.pop();
          cout << u << " "; // 输出顶点

          // 遍历所有邻接点，更新入度
          for (int v : adj[u]) {
              if (--indegree[v] == 0) {
                  q.push(v);
              }
          }
      }
      cout << endl;
  }
  };

int main()
{
Graph g(6);
g.addEdge(5, 2);
g.addEdge(5, 0);
g.addEdge(4, 0);
g.addEdge(4, 1);
g.addEdge(2, 3);
g.addEdge(3, 1);
 cout << "拓扑排序结果: ";
 g.topologicalSort();
  return 0;
  }
```

输出：

```
拓扑排序结果：4 5 2 3 1 0
```

6.6.4 关键路径（Critical Path）

关键路径是指项目管理中一个有向有环图（DAG）中，从起点到终点的最长路径，它决定了项目的最短完成时间。关键路径算法用于项目调度，帮助确定哪些任务是关键的，不能延误。

- **应用**：
- 项目调度、时间管理、资源分配等。
- **算法**：
- 利用拓扑排序和动态规划，计算最早开始时间、最晚开始时间等，最终求出关键路径。

小结

- **最小生成树**：通过 Kruskal 或 Prim 算法来构造最小生成树，应用于网络设计和优化。
- **最短路径**：通过 Dijkstra、Bellman-Ford 或 Floyd-Warshall 算法来求解最短路径问题，广泛应用于网络路由、导航系统等。
- **拓扑排序**：用于处理有向无环图（DAG）中的任务调度、依赖关系等问题。
- **关键路径**：通过拓扑排序和动态规划来求解项目调度中的关键路径问题。

第 7 章 查找

查找是数据结构中的一个重要操作，它的目的是在数据集（如数组、链表、树等）中查找某个特定元素。查找方法的选择直接影响查找的效率。不同类型的数据结构有不同的查找策略，本章将介绍几种常见的查找方法及其应用。

7.1 查找的基本概念

查找操作的目的是从一个集合中找到特定元素的位置，通常是通过比较元素来完成。根据数据结构的不同，查找算法的时间复杂度也有所不同。

- **查找算法的分类**：
    - **顺序查找**：通过逐个元素与目标元素进行比较，直至找到目标或遍历完整个集合。
    - **折半查找**：也叫二分查找，适用于有序数据，通过每次比较中间元素来缩小查找范围。
    - **分块查找**：将数据分为若干块，每次查找一个块，块内可以使用其他查找算法。

- **查找的性能评价**：
  - **时间复杂度**：表示算法在最坏情况下的执行时间。常见的时间复杂度有 O(n)、O(log n) 等。
  - **空间复杂度**：表示算法执行过程中所占用的额外存储空间。

7.2 线性表的查找

线性表是最简单的一种数据结构，常见的查找方法包括顺序查找、折半查找和分块查找。本节将分别介绍这几种方法及其实现。

7.2.1 顺序查找（Sequential Search）

顺序查找是一种最简单的查找方法，它的基本思想是从线性表的第一个元素开始，逐个与目标元素进行比较，直到找到为止。

- **算法步骤**：
  1. 从第一个元素开始遍历线性表。
  2. 将当前元素与目标元素进行比较。
  3. 如果相等，则返回该元素的位置。
  4. 如果遍历完所有元素仍未找到，则返回查找失败。
- **时间复杂度**：最坏情况下为 O(n)，其中 n 是线性表的长度。
- **C++ 代码实现**：

```
#include <iostream>
using namespace std;

// 顺序查找函数
int sequentialSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; ++i) {
        if (arr[i] == target) {
            return i;  // 返回找到的元素索引
        }
    }
    return -1;  // 未找到目标元素，返回-1
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 30;

    int result = sequentialSearch(arr, n, target);
    if (result != -1) {
        cout << "元素 " << target << " 在数组中的位置是: " << result << endl;
    } else {
        cout << "元素 " << target << " 不在数组中! " << endl;
    }

    return 0;
}
```

输出：

```
元素 30 在数组中的位置是：2
```

7.2.2 折半查找（Binary Search）

折半查找，也称为二分查找，是一种高效的查找算法，适用于已排序的线性表。它的基本思想是每次将查找区间分为两半，通过与中间元素的比较来逐步缩小查找范围。

- **算法步骤**：
  1. 如果查找区间为空，说明未找到目标元素。
  2. 计算查找区间的中间元素。
  3. 如果中间元素等于目标元素，则返回该元素的位置。
  4. 如果目标元素小于中间元素，则在左半区间继续查找。
  5. 如果目标元素大于中间元素，则在右半区间继续查找。
- **时间复杂度**：O(log n)，其中 n 是线性表的长度。
- **C++ 代码实现**：

```
#include <iostream>
using namespace std;

// 折半查找（前提：数组已排序）
int binarySearch(int arr[], int n, int target) {
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;  // 防止溢出

        if (arr[mid] == target) {
            return mid;  // 找到目标元素，返回索引
        } else if (arr[mid] < target) {
            low = mid + 1;  // 在右半部分查找
        } else {
            high = mid - 1;  // 在左半部分查找
        }
    }
    return -1;  // 查找失败
}

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 30;

    int result = binarySearch(arr, n, target);
    if (result != -1) {
        cout << "元素 " << target << " 在数组中的位置是：" << result << endl;
    } else {
        cout << "元素 " << target << " 不在数组中！" << endl;
```

```
    }

    return 0;
}
```

输出：

```
元素 30 在数组中的位置是: 2
```

7.2.3 分块查找（Block Search）

分块查找将线性表分为若干个块，每个块内可以使用其他查找方法（如顺序查找或折半查找），查找时首先确定目标元素所在的块，然后在该块内进行查找。

- **算法步骤**：
  1. 将线性表划分为若干个块，每个块的大小为 `block_size`。
  2. 使用顺序查找或折半查找确定目标元素所在的块。
  3. 在确定的块内再次进行查找。
- **时间复杂度**：最坏情况下为 O(n / m + m)，其中 n 为线性表长度，m 为块的大小。通常取 m 为常数，从而查找时间接近 O(√n)。
- **C++ 代码实现（分块查找）**：

```
#include <iostream>
#include <cmath>
using namespace std;

// 分块查找函数
int blockSearch(int arr[], int n, int target, int block_size) {
    int num_blocks = (n + block_size - 1) / block_size;  // 计算块数
    int i = 0;

    // 查找目标元素所在的块
    for (i = 0; i < num_blocks; ++i) {
        int block_start = i * block_size;
        int block_end = min((i + 1) * block_size - 1, n - 1);

        // 在块内顺序查找
        for (int j = block_start; j <= block_end; ++j) {
            if (arr[j] == target) {
                return j;
            }
        }
    }
    return -1;  // 查找失败
}

int main() {
    int arr[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 70;
```

```
    int block_size = 3;  // 块的大小为3

    int result = blockSearch(arr, n, target, block_size);
    if (result != -1) {
        cout << "元素 " << target << " 在数组中的位置是: " << result << endl;
    } else {
        cout << "元素 " << target << " 不在数组中! " << endl;
    }

    return 0;
}
```

输出:

```
元素 70 在数组中的位置是: 6
```

小结

- **顺序查找**：适用于无序数据，时间复杂度为 O(n)。
- **折半查找**：适用于有序数据，时间复杂度为 O(log n)，效率较高。
- **分块查找**：通过将数据分块，结合顺序查找或折半查找来提高查找效率，适用于大规模数据。

7.3 树表的查找

在查找操作中，树形结构（如二叉排序树、平衡二叉树、B-树、B+树）是一种有效的数据结构。它们能够提供比线性查找更高效的查找操作，尤其在需要频繁插入、删除和查找的场景下，树形结构的查找算法通常具有更好的性能。本节将介绍几种常见的树表查找方法及其实现。

7.3.1 二叉排序树（Binary Search Tree，BST）

二叉排序树是一种满足特定顺序性质的二叉树，具体要求如下：

- 每个节点的左子树只包含比当前节点小的元素。
- 每个节点的右子树只包含比当前节点大的元素。
- 左右子树的节点也分别满足上述条件。

查找算法:

1. 从根节点开始，如果目标元素等于当前节点的值，则查找成功。
2. 如果目标元素小于当前节点的值，则向左子树递归查找。
3. 如果目标元素大于当前节点的值，则向右子树递归查找。
4. 如果某一节点为空，则查找失败。

时间复杂度:

- 最坏情况下为 O(n)，即树变成了链表（树的高度为 n）。
- 最好情况下为 O(log n)，即树保持平衡。

C++ 实现:

```
#include <iostream>
using namespace std;

// 二叉树节点结构
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

// 二叉排序树类
class BinarySearchTree {
public:
    Node* root;

    BinarySearchTree() : root(nullptr) {}

    // 插入节点
    void insert(int value) {
        root = insertRec(root, value);
    }

    // 查找节点
    Node* search(int value) {
        return searchRec(root, value);
    }

private:
    // 插入的递归函数
    Node* insertRec(Node* node, int value) {
        // 如果树为空，创建新节点
        if (node == nullptr) {
            return new Node(value);
        }

        // 否则，递归地在左子树或右子树中插入
        if (value < node->data) {
            node->left = insertRec(node->left, value);
        } else if (value > node->data) {
            node->right = insertRec(node->right, value);
        }

        return node;
    }

    // 查找的递归函数
    Node* searchRec(Node* node, int value) {
        // 如果节点为空或找到目标元素，返回节点
        if (node == nullptr || node->data == value) {
            return node;
        }
```

```
        // 如果目标元素小于当前节点，递归左子树
        if (value < node->data) {
            return searchRec(node->left, value);
        }

        // 如果目标元素大于当前节点，递归右子树
        return searchRec(node->right, value);
    }
};

int main() {
    BinarySearchTree bst;

    // 插入节点
    bst.insert(50);
    bst.insert(30);
    bst.insert(20);
    bst.insert(40);
    bst.insert(70);
    bst.insert(60);
    bst.insert(80);

    // 查找节点
    int target = 40;
    Node* result = bst.search(target);
    if (result != nullptr) {
        cout << "找到节点，值为: " << result->data << endl;
    } else {
        cout << "节点 " << target << " 不在树中!" << endl;
    }

    return 0;
}
```


输出：

```
找到节点，值为: 40
```

7.3.2 平衡二叉树（AVL 树）

AVL 树是一种自平衡的二叉排序树。它的平衡因子（左子树高度减去右子树高度）始终保持在 -1、0 和 1 之间。插入或删除操作后，如果树不平衡，会通过旋转操作恢复平衡。

查找算法：与普通的二叉排序树相同，只不过在每次插入或删除节点后，需要对树进行平衡调整。

时间复杂度：插入、删除和查找操作的时间复杂度为 O(log n)。

C++ 实现（插入和查找）：

```
#include <iostream>
#include <algorithm>
```

```
using namespace std;

// 二叉树节点结构
struct Node {
    int data;
    Node* left;
    Node* right;
    int height; // 节点的高度

    Node(int value) : data(value), left(nullptr), right(nullptr), height(1) {}
};

// AVL树类
class AVLTree {
public:
    Node* root;

    AVLTree() : root(nullptr) {}

    // 插入节点
    void insert(int value) {
        root = insertRec(root, value);
    }

    // 查找节点
    Node* search(int value) {
        return searchRec(root, value);
    }

private:
    // 插入的递归函数
    Node* insertRec(Node* node, int value) {
        // 1. 执行正常的二叉搜索树插入
        if (node == nullptr) {
            return new Node(value);
        }

        if (value < node->data) {
            node->left = insertRec(node->left, value);
        } else if (value > node->data) {
            node->right = insertRec(node->right, value);
        } else {
            return node; // 如果值相等，不插入
        }

        // 2. 更新节点的高度
        node->height = 1 + max(getHeight(node->left), getHeight(node->right));

        // 3. 计算平衡因子并检查是否平衡
        int balance = getBalance(node);

        // 4. 如果不平衡，则进行相应的旋转
        if (balance > 1 && value < node->left->data) {
```

```
        return rotateRight(node); // 左左情况
    }
    if (balance < -1 && value > node->right->data) {
        return rotateLeft(node); // 右右情况
    }
    if (balance > 1 && value > node->left->data) {
        node->left = rotateLeft(node->left); // 左右情况
        return rotateRight(node);
    }
    if (balance < -1 && value < node->right->data) {
        node->right = rotateRight(node->right); // 右左情况
        return rotateLeft(node);
    }

    return node;
}

// 查找的递归函数
Node* searchRec(Node* node, int value) {
    if (node == nullptr || node->data == value) {
        return node;
    }
    if (value < node->data) {
        return searchRec(node->left, value);
    }
    return searchRec(node->right, value);
}

// 获取节点的高度
int getHeight(Node* node) {
    return node ? node->height : 0;
}

// 获取节点的平衡因子
int getBalance(Node* node) {
    return node ? getHeight(node->left) - getHeight(node->right) : 0;
}

// 右旋转
Node* rotateRight(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // 执行旋转
    x->right = y;
    y->left = T2;

    // 更新高度
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    return x;
}
```

```
    // 左旋转
    Node* rotateLeft(Node* x) {
        Node* y = x->right;
        Node* T2 = y->left;

        // 执行旋转
        y->left = x;
        x->right = T2;

        // 更新高度
        x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
        y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

        return y;
    }
};

int main() {
    AVLTree avl;

    // 插入节点
    avl.insert(30);
    avl.insert(20);
    avl.insert(40);
    avl.insert(10);
    avl.insert(5);

    // 查找节点
    int target = 20;
    Node* result = avl.search(target);
    if (result != nullptr) {
        cout << "找到节点，值为: " << result->data << endl;
    } else {
        cout << "节点 " << target << " 不在树中!" << endl;
    }

    return 0;
}
```


输出:

```
找到节点，值为: 20
```

7.3.3 B-树

B-树是一种自平衡

的多路查找树，广泛应用于数据库系统中。它适用于磁盘存储的高效查找。

B-树的特点:

- 每个节点最多有 m 个子节点。
- 所有的叶子节点都在同一层。
- 内部节点的关键字按照升序排列。

7.3.4 B+ 树

B+ 树是 B-树的变种，主要区别在于它的所有数据都保存在叶子节点中，内节点只保存索引。它通常用于数据库和文件系统中，用来优化查找性能。

以上为树表查找的几种常见方法。每种树结构的查找性能不同，B+ 树尤其适用于大量数据的磁盘存储和查找场景，而 AVL 树和二叉排序树则适用于内存中的数据查找。

7.4 散列表的查找

散列表（Hash Table）是一种基于哈希函数的查找数据结构，它通过将关键字映射到一个固定大小的数组下标来进行数据的存储和查找。散列表的查找时间复杂度通常为 O(1)O(1)，但在出现冲突时，查找的时间复杂度可能会增加。散列表在需要频繁查找操作的场景中表现出色，特别是在处理大量数据时。

本节将详细介绍散列表的基本概念、哈希函数的构造方法、处理冲突的方法以及散列表的查找操作。

7.4.1 散列表的基本概念

散列表通过哈希函数将关键字映射到一个固定大小的数组中。每个位置可以存储一个元素，当多个元素映射到同一个位置时，发生**冲突**。散列表解决冲突的方法有多种，包括链式地址法、开放地址法等。

散列表的基本概念：

- **哈希函数（Hash Function）**：将输入的关键字映射到数组索引的函数。
- **哈希表（Hash Table）**：由一组数组和哈希函数组成的存储结构。
- **冲突**：当多个不同的元素通过哈希函数映射到相同的数组位置时，就发生了冲突。
- **负载因子（Load Factor）**：表示哈希表已使用空间与总空间的比例。负载因子高会导致冲突的发生频率增加。

7.4.2 散列函数的构造方法

哈希函数的目的是将输入的关键字均匀地映射到哈希表的各个槽位上。一个好的哈希函数能够减少冲突的发生。以下是常见的几种哈希函数构造方法：

1. **除法法**：使用关键字对一个素数取余得到数组的索引。常用的素数是质数，因为质数能够有效减少冲突。

   哈希函数的形式为：

   h(k)=kmod ph(k) = k \mod p

   其中 kk 是关键字，pp 是一个素数，\mod 表示取余操作。

2. **乘法法**：使用关键字乘以一个常数 AA，然后取整数部分的余数。常常取 A=(5−1)/2A = (\sqrt{5} - 1) / 2，即黄金分割数的倒数，以优化哈希函数的分布。

   哈希函数的形式为：

   h(k)=⌊m(kAmod 1)⌋h(k) = \lfloor m(kA \mod 1) \rfloor

其中 mm 是哈希表的大小，⌊x⌋\lfloor x \rfloor 表示取整操作，mod 1\mod 1 表示取小数部分。

3. **平方法**：对关键字进行平方运算后，取中间的若干位作为哈希值。

7.4.3 处理冲突的方法

当多个关键字通过哈希函数映射到同一位置时，会发生**冲突**。处理冲突的常见方法有两种：**链式地址法**和**开放地址法**。

1. 链式地址法

链式地址法通过在哈希表的每个位置使用一个链表来存储所有映射到该位置的元素。如果多个元素的哈希值相同，它们将存储在同一个链表中。

链式地址法的优点：

- 插入和删除操作简单。
- 无需重新哈希数组大小，适应性强。

缺点：

- 在极端情况下（例如所有元素都映射到同一位置），查找时间复杂度可能变为 O(n)O(n)。

2. 开放地址法

开放地址法通过查找表中其他空闲的位置来存储冲突的元素，常见的开放地址法处理冲突的方式包括：

- **线性探测**：如果当前位置已被占用，则依次检查下一个位置，直到找到空槽。
- **二次探测**：在发生冲突时，使用平方函数跳跃到下一个位置。
- **双重哈希**：使用第二个哈希函数来确定冲突后的跳跃步长。

开放地址法的优点：

- 不需要额外的存储空间（不需要链表）。
- 查找较为高效，尤其在负载因子较低时。

缺点：

- 插入、删除操作可能需要较多的查找步骤。
- 当负载因子增加时，性能下降较为显著。

7.4.4 散列表的查找

在散列表中查找一个元素的过程：

1. 使用哈希函数计算目标元素的哈希值，得到对应的槽位。
2. 在该槽位检查是否有目标元素。
   - 如果没有冲突，直接返回该元素。
   - 如果发生冲突，使用冲突解决方法（链式地址法或开放地址法）继续查找。

时间复杂度：

- 最坏情况下，查找时间复杂度为 O(n)O(n)，即所有元素都映射到同一位置。
- 平均情况下，查找时间复杂度为 O(1)O(1)，如果哈希函数和冲突解决方法设计良好。

C++ 实现（使用链式地址法处理冲突）：

```
#include <iostream>
#include <list>
using namespace std;

class HashTable {
public:
    HashTable(int size) : table(size) {}

    // 哈希函数
    int hash(int key) {
        return key % table.size(); // 使用除法法
    }

    // 插入元素
    void insert(int key) {
        int index = hash(key);
        table[index].push_back(key); // 将元素插入到链表中
    }

    // 查找元素
    bool search(int key) {
        int index = hash(key);
        for (int item : table[index]) {
            if (item == key) {
                return true; // 找到元素
            }
        }
        return false; // 没有找到元素
    }

    // 删除元素
    void remove(int key) {
        int index = hash(key);
        table[index].remove(key); // 从链表中删除元素
    }

private:
    vector<list<int>> table; // 使用链表数组作为哈希表
};

int main() {
    HashTable ht(10); // 创建一个大小为 10 的哈希表

    ht.insert(10);
    ht.insert(20);
    ht.insert(30);
    ht.insert(15);

    cout << "查找 20: " << (ht.search(20) ? "找到" : "未找到") << endl;
    cout << "查找 40: " << (ht.search(40) ? "找到" : "未找到") << endl;
```

```
    ht.remove(20);
    cout << "查找 20: " << (ht.search(20) ? "找到" : "未找到") << endl;

    return 0;
}
```

输出：

```
查找 20: 找到
查找 40: 未找到
查找 20: 未找到
```

总结

散列表是一种非常高效的查找数据结构，特别是在需要频繁进行查找操作时。通过良好的哈希函数设计和有效的冲突处理方法，散列表能够在大多数情况下实现 O(1)O(1) 的查找效率。链式地址法和开放地址法是两种常见的处理冲突的方法，适用于不同的应用场景。在实际使用中，选择合适的哈希函数和冲突解决策略对于散列表的性能至关重要。

第 8 章 排序

排序是计算机科学中的基本问题之一，其目的是将一个无序的元素集合按照某种顺序排列。排序广泛应用于数据库管理、搜索算法、图形图像处理等各个领域。高效的排序算法不仅能提高程序的性能，还能优化整体系统的响应速度。

本章主要介绍排序的基本概念、内部排序方法的分类、待排序记录的存储方式，以及如何评价排序算法的效率。

8.1 基本概念和排序方法概述

排序是将一个序列中的元素按照某种特定的顺序（如升序或降序）排列的过程。排序操作有多种方法，选择合适的排序算法对于提高程序的性能至关重要。

8.1.1 排序的基本概念

排序的基本目标是将一个无序的记录序列，按照关键字的大小进行排列。排序可以分为两类：

- **内部排序**：所有待排序的记录都可以存储在内存中，排序过程完全在内存中进行。
- **外部排序**：待排序的记录量大到无法全部放入内存时，排序需要利用外部存储设备（如磁盘）来辅助排序。

排序的目标是：根据给定的比较规则，将元素按升序或降序排列。排序操作的结果应该使得任何一个元素都比它后面的元素要小（或大，视排序方式而定）。

8.1.2 内部排序方法的分类

内部排序方法主要分为以下几类：

1. **比较排序**：比较排序是通过比较元素之间的大小来决定元素的顺序。这类排序算法的时间复杂度通常是 O(nlogn)O(n \log n) 或更高。常见的比较排序算法包括：
    - 冒泡排序
    - 插入排序

- 选择排序
- 快速排序
- 归并排序
- 堆排序

2. **非比较排序**：非比较排序不通过比较元素大小来排序，而是利用其他方法（如计数、基数等）来实现排序。常见的非比较排序算法包括：
   - 计数排序
   - 桶排序
   - 基数排序
3. **稳定排序与不稳定排序**：
   - **稳定排序**：如果两个元素相等，排序后它们的相对位置不会改变。常见的稳定排序算法有插入排序、归并排序和计数排序。
   - **不稳定排序**：相等元素的相对位置可能会发生变化。常见的不稳定排序算法有快速排序和选择排序。

8.1.3 待排序记录的存储方式

排序的效率与待排序数据的存储方式密切相关。常见的存储方式有：

- **顺序存储**：将数据存储在连续的内存空间中，适合对数组或链表进行排序。常见的排序算法（如冒泡排序、快速排序）通常使用顺序存储。
- **链式存储**：将数据存储在链表等数据结构中，这种存储方式常常用在一些特定的排序算法中（如归并排序中的链表实现）。

不同的存储方式会对排序算法的实现和效率产生影响。例如，链式存储的排序可能需要额外的指针操作来维护链表的连接，而顺序存储的排序可以直接通过数组下标进行访问。

8.1.4 排序算法效率的评价指标

评估排序算法的效率通常使用以下几个指标：

1. **时间复杂度**：时间复杂度表示算法随着数据规模 nn 增长，所需要的时间增长速度。排序算法的时间复杂度通常分为三种情况：
   - **最优时间复杂度**：算法在最好情况下的表现。
   - **最坏时间复杂度**：算法在最坏情况下的表现。
   - **平均时间复杂度**：算法在平均情况下的表现。

   对于比较排序算法，最好的时间复杂度通常是 O(nlogn)O(n \log n)，最差情况下可能会退化为 O(n2)O(n^2)，如冒泡排序和插入排序。
2. **空间复杂度**：空间复杂度表示排序算法所需要的额外空间。对于一些排序算法，如归并排序和快速排序，可能需要额外的存储空间，而其他算法（如冒泡排序、插入排序）则可以在原地排序，空间复杂度为 O(1)O(1)。
3. **稳定性**：稳定性是指当两个元素相等时，它们在排序后相对位置是否会发生改变。稳定性在一些应用中非常重要，如在多重排序中，稳定的排序算法可以确保之前排序的顺序保持不变。
4. **适应性**：一些排序算法能够在待排序数据本身已接近排序状态时，提供更好的性能。例如，插入排序在数组接近有序时表现出 O(n)O(n) 的时间复杂度。

总结

排序算法是计算机科学中的基础技术，选择合适的排序算法能够显著提高程序的效率。在本节中，我们讨论了排序的基本概念、常见的排序方法及其分类、待排序记录的存储方式，以及排序算法效率的评价指标。选择正确的排序算法时，需要根据数据的规模、存储方式、以及应用场景来决定。

8.2 插入排序

插入排序是一种简单的排序算法，其工作原理类似于我们整理扑克牌的过程。它将数据分为“已排序部分”和“未排序部分”，每次从未排序部分取出一个元素，插入到已排序部分的合适位置，直到所有元素都有序。

插入排序有不同的变种，常见的包括**直接插入排序**、**折半插入排序**和**希尔排序**。

8.2.1 直接插入排序

直接插入排序是插入排序的一种最基本形式，它将未排序的元素逐个与已排序部分进行比较，将其插入到合适的位置。其核心思想是不断将当前元素插入到已经排好序的部分。

算法步骤：

1. 从第二个元素开始，将其与前面的元素进行比较。
2. 如果当前元素小于前面的元素，则将前面的元素后移一位，为当前元素腾出位置。
3. 直到找到合适的位置，将当前元素插入。
4. 对所有元素重复以上步骤，直到所有元素排序完成。

时间复杂度：

- 最坏情况下：如果数组是逆序排列，时间复杂度为 O(n2)O(n^2)。
- 最好情况下：如果数组已经有序，时间复杂度为 O(n)O(n)。
- 平均情况下：时间复杂度为 O(n2)O(n^2)。

代码实现（C++）：

```
#include <iostream>
using namespace std;

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];  // 当前元素
        int j = i - 1;

        // 将大于 key 的元素移到一个位置后
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;  // 将 key 插入到正确位置
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    insertionSort(arr, n);
```

```
    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

输出:

```
排序后的数组: 5 6 11 12 13
```

8.2.2 折半插入排序

折半插入排序是对直接插入排序的一种优化。直接插入排序每次比较时都从数组的头部开始，时间复杂度较高。而折半插入排序通过使用二分查找法找到合适的位置，从而减少比较次数。

算法步骤:

1. 对每个待插入的元素，使用二分查找在已排序部分找到合适的位置。
2. 在找到的位置插入元素，并将已排序部分的元素后移。

时间复杂度:

- 最坏情况下：时间复杂度为 O(n2)O(n^2)，和直接插入排序相同，因为元素的移动次数依然是 O(n)O(n)。
- 平均情况下：时间复杂度为 O(n2)O(n^2)。
- 但是相比于直接插入排序，折半插入排序在比较的次数上有所减少，提升了效率。

代码实现（C++）：

```
#include <iostream>
using namespace std;

// 使用二分查找找到插入的位置
int binarySearch(int arr[], int key, int low, int high) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (key == arr[mid]) return mid + 1;
        if (key < arr[mid]) high = mid - 1;
        else low = mid + 1;
    }
    return low;
}

void binaryInsertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int low = 0, high = i - 1;

        // 使用二分查找找到插入的位置
        int pos = binarySearch(arr, key, low, high);
```

```
        // 将元素后移
        for (int j = i - 1; j >= pos; j--) {
            arr[j + 1] = arr[j];
        }
        arr[pos] = key;  // 插入元素
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    binaryInsertionSort(arr, n);

    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

输出:

```
排序后的数组: 5 6 11 12 13
```

8.2.3 希尔排序

希尔排序是插入排序的一种优化版本，基本思想是将整个数组分成若干子序列（通过一个步长序列），对每个子序列分别进行插入排序。随着步长逐渐缩小，排序逐渐细化，最终变成直接插入排序。

算法步骤:

1. 选择一个步长序列（例如：`n/2, n/4, ..., 1`），将待排序的数组按步长划分为若干子序列。
2. 对每个子序列执行插入排序，步长逐步减小，直到步长为 1 时，再对整个数组执行一次插入排序。

时间复杂度:

- 最坏情况下：时间复杂度依赖于步长序列的选择，通常是 O(n3/2)O(n^{3/2})。
- 最好情况下：时间复杂度为 O(nlogn)O(n \log n)（取决于步长序列）。

代码实现（C++）:

```
#include <iostream>
using namespace std;

void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int key = arr[i];
            int j = i;
```

```
            while (j >= gap && arr[j - gap] > key) {
                arr[j] = arr[j - gap];
                j -= gap;
            }
            arr[j] = key;
        }
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    shellSort(arr, n);

    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

输出:

```
排序后的数组: 5 6 11 12 13
```

总结

- **直接插入排序**：适用于小规模数据，平均时间复杂度为 O(n2)O(n^2)，当数据基本有序时，效率较高。
- **折半插入排序**：通过二分查找减少比较次数，但元素移动次数和直接插入排序相同，效率依然较低。
- **希尔排序**：通过缩小步长改善了插入排序的性能，适合中等规模的数据。时间复杂度与步长序列密切相关，最优情况下能达到 O(nlogn)O(n \log n)。

插入排序及其变种是简单而直观的排序算法，适合用于小规模的数据排序，或当数据接近有序时，表现出较高的效率。

8.3 交换排序

交换排序是一类通过交换元素位置来实现排序的算法。常见的交换排序算法包括**冒泡排序**和**快速排序**。这些算法通常具有 O(n2)O(n^2) 的时间复杂度，但通过一些优化，快速排序的性能可以大大提升，成为高效的排序方法。

8.3.1 冒泡排序

冒泡排序是一种简单的排序算法，它通过重复地交换相邻的逆序元素来将较大的元素“冒泡”到数组的末端。算法的核心思想是每次遍历数组，将最大（或最小）的元素放到已排序部分的最后。

算法步骤:

1. 从数组的第一个元素开始，依次比较相邻的两个元素。

2. 如果当前元素大于下一个元素，则交换这两个元素的位置。
3. 每次遍历完成后，最大的元素会被移到数组的末尾。
4. 重复以上过程，直到整个数组排序完成。

时间复杂度：

- 最坏情况下：时间复杂度为 O(n2)O(n^2)，当数组逆序时，冒泡排序会进行最多 n−1n-1 次的交换操作。
- 最好情况下：时间复杂度为 O(n)O(n)，如果数组已经有序，则只需要进行一次遍历。
- 平均情况下：时间复杂度为 O(n2)O(n^2)。

代码实现（C++）：

```
#include <iostream>
using namespace std;

void bubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n-1; i++) {
        swapped = false;
        // 从前往后遍历数组
        for (int j = 0; j < n-i-1; j++) {
            // 如果当前元素比下一个元素大，则交换
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
                swapped = true;
            }
        }
        // 如果没有发生交换，说明数组已经有序，可以提前结束排序
        if (!swapped) {
            break;
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);

    bubbleSort(arr, n);

    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```


输出：

```
排序后的数组: 11 12 22 25 34 64 90
```

8.3.2 快速排序

快速排序是一种分治算法，它的基本思想是选择一个“基准”元素，然后将比基准元素小的元素放到左边，比基准元素大的元素放到右边。接着，对左右两个子数组分别进行快速排序。这个过程递归地进行，直到所有子数组都排好序。

算法步骤：

1. 选择一个基准元素（通常选择数组的第一个、最后一个或随机一个元素）。
2. 将数组重新排列，使得所有比基准小的元素排在基准的左边，所有比基准大的元素排在右边。
3. 递归地对基准左侧和右侧的子数组进行快速排序。

时间复杂度：

- 最坏情况下：时间复杂度为 O(n2)O(n^2)，当数组已经是有序或逆序时，分割后的子数组可能不均匀，导致递归深度过大。
- 最好情况下：时间复杂度为 O(nlogn)O(n \log n)，当每次划分的子数组都均匀时，递归树的深度为 O(logn)O(\log n)。
- 平均情况下：时间复杂度为 O(nlogn)O(n \log n)，对于随机输入，快速排序通常表现出较好的性能。

代码实现（C++）：

```
#include <iostream>
using namespace std;

// 交换两个元素
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// 分割函数，将数组分为两部分，小于基准元素的部分和大于基准元素的部分
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // 选择最后一个元素作为基准
    int i = (low - 1);  // i 是小于基准元素的区域的最后一个元素的索引

    for (int j = low; j < high; j++) {
        // 如果当前元素小于等于基准元素，则将其与小于区域的下一个元素交换
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    // 将基准元素放到正确的位置
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// 快速排序函数
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // 找到基准元素的正确位置
```

```
        int pi = partition(arr, low, high);

        // 递归地对基准元素左侧和右侧进行快速排序
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);

    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

输出:

```
排序后的数组: 1 5 7 8 9 10
```

总结

- **冒泡排序**：是一种简单的交换排序算法，适用于小规模数据排序，最坏情况下时间复杂度为 O(n2)O(n^2)。
- **快速排序**：是一种分治排序算法，具有较高的效率，平均时间复杂度为 O(nlogn)O(n \log n)，但在最坏情况下会退化为 O(n2)O(n^2)。快速排序是大多数实际应用中最常用的排序算法之一。

尽管冒泡排序简单易懂，但其效率较低，主要用于教学和小规模数据排序；而快速排序在大规模数据中表现出优异的性能，是最常用的排序算法之一。

8.4 选择排序

选择排序是一种简单的排序算法，它的基本思想是每次从待排序的部分中选择最小（或最大）元素，将其放到已排序部分的末尾。通过不断选择未排序部分的最小值进行交换，最终将整个数组排好序。

常见的选择排序算法有**简单选择排序**、**树形选择排序**和**堆排序**。

8.4.1 简单选择排序

简单选择排序是一种直接的排序方法，每一趟都从待排序的部分中找到最小的元素，然后与未排序部分的第一个元素交换，重复此过程，直到数组排序完成。

算法步骤:

1. 从数组的第一个元素开始，遍历整个数组，找到最小的元素。

2. 将最小的元素与当前遍历位置的元素交换。
3. 然后从剩余部分重复以上步骤。
4. 直到整个数组排好序。

时间复杂度：

- 最坏情况下：时间复杂度为 O(n2)O(n^2)。
- 最好情况下：时间复杂度为 O(n2)O(n^2)，因为每一趟都要进行遍历。
- 平均情况下：时间复杂度为 O(n2)O(n^2)，适用于小规模数据。

代码实现（C++）：

```
#include <iostream>
using namespace std;

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;  // 假设当前位置的元素是最小的
        for (int j = i + 1; j < n; j++) {
            // 找到当前未排序部分中的最小元素
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // 如果最小元素不是当前位置的元素，进行交换
        if (minIndex != i) {
            swap(arr[i], arr[minIndex]);
        }
    }
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

    selectionSort(arr, n);

    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```


输出：

```
排序后的数组: 11 12 22 25 34 64 90
```

8.4.2 树形选择排序

树形选择排序是一种通过构建完全二叉树来优化选择排序的算法。在树形选择排序中，首先将所有元素构建成一颗完全二叉树，然后依次从树的根节点选择最小（或最大）元素进行交换。其思想类似于选择排序，但它使用树形结构来进行选择操作，减少了元素的比较次数。

算法步骤：

1. 构建一颗完全二叉树，将待排序的元素插入到树的叶子节点。
2. 将树从最后一个非叶子节点开始，进行堆化操作，使其满足堆的性质。
3. 将堆的根节点与最后一个节点交换，并重新堆化，重复此过程，直到堆为空。

时间复杂度：

- 最坏情况下：时间复杂度为 O(nlogn)O(n \log n)，因为每次堆化操作的时间复杂度为 O(logn)O(\log n)。
- 最好情况下：时间复杂度为 O(nlogn)O(n \log n)。
- 平均情况下：时间复杂度为 O(nlogn)O(n \log n)，比简单选择排序要高效。

代码实现（C++）：

```
#include <iostream>
using namespace std;

// 堆化操作
void heapify(int arr[], int n, int i) {
    int largest = i;  // 假设根节点是最大值
    int left = 2 * i + 1;  // 左子节点
    int right = 2 * i + 2;  // 右子节点

    // 如果左子节点更大
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // 如果右子节点更大
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // 如果根节点不是最大值，交换并递归堆化
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

// 树形选择排序（堆排序）
void heapSort(int arr[], int n) {
    // 建立最大堆
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // 逐步从堆中取出元素
    for (int i = n - 1; i > 0; i--) {
        // 将堆顶元素与堆的最后一个元素交换
```

```
        swap(arr[0], arr[i]);

        // 重新堆化
        heapify(arr, i, 0);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

输出:

```
排序后的数组: 5 6 7 11 12 13
```

8.4.3 堆排序

堆排序是一种树形选择排序，基于完全二叉树的性质。堆排序首先将待排序数组构建成一个大根堆（或小根堆），然后将堆顶元素（最大或最小）与最后一个元素交换，并进行堆化操作，逐步缩小堆的范围，直到排序完成。

堆排序利用了**堆**这一数据结构，确保了每个父节点都大于（或小于）其子节点，从而可以在 O(logn)O(\log n) 的时间内进行堆化。

算法步骤:

1. 将数组构建为一个大根堆（对于升序排序），即每个节点都大于或等于其子节点。
2. 将堆顶元素（最大元素）与堆的最后一个元素交换，将最大元素放到正确的位置。
3. 对剩下的 n−1n-1 个元素进行堆化操作，保持堆的性质。
4. 重复以上过程，直到整个数组排序完成。

时间复杂度:

- 最坏情况下：时间复杂度为 O(nlogn)O(n \log n)。
- 最好情况下：时间复杂度为 O(nlogn)O(n \log n)。
- 平均情况下：时间复杂度为 O(nlogn)O(n \log n)。

代码实现（C++）：

```
#include <iostream>
using namespace std;
```

```
// 堆化操作
void heapify(int arr[], int n, int i) {
    int largest = i;  // 假设根节点是最大值
    int left = 2 * i + 1;  // 左子节点
    int right = 2 * i + 2;  // 右子节点

    // 如果左子节点更大
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // 如果右子节点更大
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // 如果根节点不是最大值，交换并递归堆化
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

// 堆排序
void heapSort(int arr[], int n) {
    // 建立最大堆
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // 逐步从堆中取出元素
    for (int i = n - 1; i > 0; i--) {
        // 将堆顶元素与堆的最后一个元素交换
        swap(arr[0], arr[i]);

        // 重新堆化
        heapify(arr, i, 0);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
```

```
    return 0;
}
```

输出

：

```
排序后的数组: 5 6 7 11 12 13
```

总结

- **简单选择排序**：每次选择最小元素进行交换，时间复杂度为 O(n2)O(n^2)，适用于小规模数据排序。
- **树形选择排序**：通过构建堆来优化选择排序，时间复杂度为 O(nlogn)O(n \log n)。
- **堆排序**：基于堆的选择排序算法，时间复杂度为 O(nlogn)O(n \log n)，是一种效率较高的排序方法，适用于大规模数据排序。

堆排序在时间效率上较为优秀，且具有 O(nlogn)O(n \log n) 的最坏时间复杂度，但需要额外的空间来存储堆。

8.5 归并排序

归并排序（Merge Sort）是一种典型的分治法（Divide and Conquer）排序算法。它将一个大的问题分解为多个小的子问题，并递归地解决每个子问题，最后将解决方案合并。

基本思想：

1. 将待排序的数组分成两半，递归地对两半分别进行排序。
2. 合并两个已排序的子数组，得到最终的排序数组。

归并排序的关键在于**合并操作**，即将两个已排序的部分合并成一个有序数组。

时间复杂度：

- 最坏、最好和平均情况下的时间复杂度均为 O(nlogn)O(n \log n)。
- 空间复杂度为 O(n)O(n)，因为归并排序需要额外的空间来存储合并过程中的数组。

代码实现（C++）：

```
#include <iostream>
using namespace std;

// 合并两个子数组
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    // 将数据拷贝到临时数组 L[] 和 R[]
    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }
    for (int i = 0; i < n2; i++) {
```

```
        R[i] = arr[mid + 1 + i];
    }

    int i = 0, j = 0, k = left;

    // 合并两个子数组
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // 将 L[] 中剩余的元素拷贝到 arr[]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // 将 R[] 中剩余的元素拷贝到 arr[]
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// 归并排序的递归实现
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);  // 排序左半部分
        mergeSort(arr, mid + 1, right);  // 排序右半部分

        merge(arr, left, mid, right);  // 合并两个已排序的子数组
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, n - 1);

    cout << "排序后的数组: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
```

```
    }
    cout << endl;

    return 0;
}
```

输出：

```
排序后的数组: 5 6 7 11 12 13
```

归并排序的优点：

- **稳定性**：归并排序是稳定排序，它不会改变相等元素的顺序。
- **时间复杂度**：最坏、最好和平均时间复杂度都为 O(nlogn)O(n \log n)，相对其他排序算法，性能稳定。
- **适合大数据**：因为其稳定的 O(nlogn)O(n \log n) 时间复杂度，它适合于排序大数据。

8.6 基数排序

基数排序（Radix Sort）是一种非比较型的整数排序算法，它通过将数值按位数拆分进行排序，从低位到高位逐步排序，最终得到一个有序的数组。

基数排序利用了整数的每一位（从低位到高位）进行排序，并在每一位的排序中使用稳定的排序算法（通常是计数排序）。它的效率非常高，但只适用于整数或者能够转化为整数的场合。

基本思想：

1. 从最低位开始，对所有数进行一次排序。
2. 然后按次序逐步处理更高的位，直到最高位。
3. 每次排序使用的是稳定排序（如计数排序），确保相同数字的位置不会改变。

时间复杂度：

- 如果有 nn 个数字，每个数字的位数为 dd，则时间复杂度为 O(n·d)O(n \cdot d)。
- 对于固定范围内的整数，基数排序是非常高效的。

8.6.1 多关键字的排序

多关键字排序是指当一个对象由多个属性组成时，按照多个属性进行排序。基数排序可以作为多关键字排序的基础，通过对每个关键字进行逐位排序，最终得到多关键字排序结果。

时间复杂度：与基数排序相同，时间复杂度为 O(n·d)O(n \cdot d)，其中 nn 是数据项的个数，dd 是关键字的位数。

8.6.2 链式基数排序

链式基数排序是在基数排序中使用**链表**作为桶的存储结构。在每一轮的排序中，数字根据当前位数的值被分配到不同的桶中。链式基数排序使用链表来存储桶中的元素，从而避免了元素的冲突。

算法步骤：

1. 为每个可能的位数（从0到9）创建一个链表桶。
2. 根据每个数字当前位的值，将数字插入到相应的链表桶中。
3. 从链表桶中按顺序取出元素，重新组成数组，进行下一轮排序。

时间复杂度：和传统基数排序类似，时间复杂度为 O(n·d)O(n \cdot d)。

总结

- **归并排序**：基于分治法，通过将数组递归地分割并合并，具有稳定的 O(nlogn)O(n \log n) 时间复杂度，适用于大规模数据排序。
- **基数排序**：适用于整数排序，通过逐位排序来实现，时间复杂度为 O(n·d)O(n \cdot d)，适合大量数据且数字位数较少的情况。

8.7 外部排序

外部排序（External Sorting）是当待排序的数据量大到不能完全放入内存时，采用的一种排序方法。由于数据量超出了内存容量，外部排序算法需要使用外部存储（如磁盘）来辅助完成排序。外部排序的基本目标是减少磁盘 I/O 操作，充分利用内存。

外部排序的核心思想是将大文件分割成小块，分别对每个小块进行排序，然后通过某种合并方式将这些排序好的小块合并成一个完整的排序文件。常见的外部排序算法包括多路归并和置换-选择排序。

8.7.1 外部排序的基本方法

外部排序的基本方法通常包括以下几个步骤：

1. **分割**：将大文件分成多个小块，并将每一块加载到内存中。
2. **排序**：对每一块进行内存排序，常用的排序算法有快速排序、归并排序等。
3. **归并**：使用多路归并算法将所有已排序的小块合并成一个大的排序文件。

核心挑战：外部排序的主要挑战在于磁盘 I/O 操作的效率，因为磁盘访问远比内存访问要慢得多。因此，外部排序需要尽量减少磁盘 I/O 的次数。

常用的外部排序技术：

- **多路归并（K-way Merge）**：通过优先队列（最小堆）等数据结构，实现对多个有序文件的合并。
- **置换-选择排序**：一种逐步从磁盘读取数据，并逐步将最小（或最大）值放到内存中排序的算法。

8.7.2 多路平衡归并的实现

多路平衡归并（K-way Merge）是一种外部排序算法，广泛应用于大数据处理。其基本思想是利用最小堆来实现多个已排序文件的合并。

算法步骤：

1. **初始化**：将每个已排序文件的第一个元素放入最小堆中，堆的大小为文件数目。
2. **归并过程**：每次从堆中取出最小的元素，将该元素写入最终的排序文件，并从相应的输入文件中读取下一个元素，放入堆中。
3. **重复**：继续从堆中取出最小元素，直到所有文件都归并完毕。

代码实现（C++）：

```
#include <iostream>
#include <queue>
#include <vector>
```

```
using namespace std;

// 用来比较的结构体
struct Element {
    int value;  // 元素的值
    int fileIndex;  // 元素所在的文件编号
    int indexInFile;  // 元素在文件中的位置

    // 用于优先队列的比较函数，确保最小元素优先
    bool operator>(const Element& other) const {
        return value > other.value;
    }
};

// 外部归并的多路平衡归并实现
void kWayMerge(vector<vector<int>>& sortedFiles) {
    priority_queue<Element, vector<Element>, greater<Element>> minHeap;
    vector<int> indices(sortedFiles.size(), 0);  // 各个文件的当前索引

    // 初始化堆
    for (int i = 0; i < sortedFiles.size(); i++) {
        if (sortedFiles[i].size() > 0) {
            minHeap.push({sortedFiles[i][0], i, 0});
        }
    }

    // 执行归并过程
    while (!minHeap.empty()) {
        Element current = minHeap.top();
        minHeap.pop();

        // 输出最小元素（实际应用中是写入磁盘）
        cout << current.value << " ";

        // 如果当前文件还有更多元素，则将下一个元素加入堆
        if (current.indexInFile + 1 < sortedFiles[current.fileIndex].size()) {
            int nextValue = sortedFiles[current.fileIndex][current.indexInFile + 1];
            minHeap.push({nextValue, current.fileIndex, current.indexInFile + 1});
        }
    }
    cout << endl;
}

int main() {
    vector<vector<int>> sortedFiles = {
        {1, 5, 9},   // 文件1
        {2, 6, 10},  // 文件2
        {3, 7, 11},  // 文件3
        {4, 8, 12}   // 文件4
    };

    kWayMerge(sortedFiles);  // 调用多路归并函数
```

```
    return 0;
}
```

输出：

```
1 2 3 4 5 6 7 8 9 10 11 12
```

解释：

- 程序首先将每个文件的首元素放入优先队列（最小堆）。
- 然后通过不断从堆中取出最小的元素，合并所有文件，直到所有文件的元素都被处理完。

8.7.3 置换-选择排序

置换-选择排序（Replacement Selection Sort）是外部排序的另一种常见方法，特别适用于对大量数据的排序。

算法步骤：

1. **分配内存**：首先将大文件的部分数据加载到内存中。
2. **选择排序**：对内存中的数据进行排序，选取最小或最大的元素进行替换。每次选取一个元素，放入已排序的部分，同时从磁盘读取下一个元素来填补内存中的空位。
3. **替换**：通过替换选择排序中最小的或最大的元素，逐步将数据排序。

特点：

- 每次读取的数据量是固定的，不会超过内存的大小。
- 不需要外部归并的操作，但每次替换时需要适当的缓存和处理。

8.7.4 最佳归并树

最佳归并树（Optimal Merge Tree）是一种用于外部排序的优化方法，旨在优化多路归并的归并过程。它通过构建一棵最优的二叉树来决定归并的顺序。

构建步骤：

1. 将所有的文件视为叶节点，构建一棵最优的二叉树。
2. 每次选择合并代价最小的两颗子树进行归并，直到只剩下根节点。

最优归并树的目标：减少磁盘 I/O 操作次数。通过选择合并最小的子树，能够降低总的归并成本。

总结

- **外部排序**：解决了无法将所有数据加载到内存中的问题，常用方法包括多路归并和置换-选择排序。
- **多路平衡归并**：通过最小堆实现多个已排序文件的合并，时间复杂度为 O(nlogk)O(n \log k)，其中 nn 是总数据量，kk 是文件数量。
- **置换-选择排序**：利用选择排序逐步填补内存，适用于外部排序中的数据替换操作。
- **最佳归并树**：优化归并过程，减少磁盘 I/O 操作，通过构建最优的合并树来提高效率。

外部排序的核心在于减少磁盘 I/O 操作，通过合理地设计算法和存储结构，可以显著提高大数据量排序的效率。